A DEVELOPER SERVICES COMPANY

developmentor®

DEVELOP.COM

developmentor®
A DEVELOPER SERVICES COMPANY

DEVELOP.COM

Guerrilla .NET

Student Manual

Book 2

developmentor

DEVELOP.COM

A DEVELOPER SERVICES COMPANY

developmentor®

DEVELOP.COM

[redacted]

TODO:
some surfing
on indigo to DB
mono - linux / apache impl.
of asp.Net.

Guerrilla .NET
Student Manual
Book 2

For information about other DevelopMentor resources,
please visit: www.develop.com or,
phone us at:

800.669.1932	In the US
310.543.1716	corporate office direct
08000.562.265	within the UK
+44.1242.525.108	within Europe

FW100_4.22.02

Module 11

Remoting

The .NET remoting infrastructure extends cross-AppDomain activation and method invocation to the wire. Revolving around the CLR's type system, .NET remoting provides seemless support for CLR-to-CLR distributed object computing.

After completing this module, you should be able to:

- understand the CLR remoting infrastructure
- understand the various activation semantics provided by the CLR
- understand how .NET remoting relates to Web Services

Remoting

The CLR provides considerable support for building distributed components. A variety of activation semantics facilitate intranet and internet-based deployment scenarios.

The Problem Space

Distributed component designs require extra effort

- Surrogate process needed to host object
- Assembly and type name necessary for activation, but no longer sufficient
- Host machine, process, context of object instance must be known
- Firewalls must be considered
- Marshaling performance must be considered
- Over-the-wire interoperability must be considered
- Variety of activation semantics possible

Method remoting overview - client context

All cross-context invocation is message based

- Client invokes method on a transparent proxy (TP)
- TP builds message, passes to real proxy (RP)
- RP passes message through 0 or more message sinks (interceptors) in client context
- Terminating sink in client context sends message into channel
- Channel uses formatter to prepare message for transit

This is a behind-the-scenes description of what happens when you know that the type being programmed against resides in another context. Consider the following client code:

```
using System;

class App
{
  static void Main()
  {
    Calc c = new Calc();
    int result = c.Add(2, 2);
    Console.WriteLine("2 + 2 = {0}", result);
  }
}
```

Assuming the calculator instance resides in another context, then the variable `c` refers to a transparent proxy. It is termed a "proxy" because it's a local representation of an object that resides somewhere else. It is termed "transparent" because, from a type perspective, the type hierarchy of the proxy is exactly the same as the type hierarchy of the actual object. In other words, the proxy looks and feels just like a Calc object to the programmer.

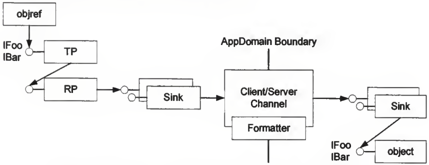
In this scenario then, the parameters passed to the `Add` method are delivered to the transparent proxy on the callstack. The transparent proxy uses the parameters on the callstack to build a message packet. This packet is then forward to the "real proxy", whose job it is to do the real work of getting the message to the target context. Along the way, the message might pass through one or more interceptors (sinks). Each sink has the opportunity to modify and/or add information to the message as it travels along its route to the target context.

Method remoting overview - server context

All cross-context invocation is message based

- Server channel receives message
- Channel uses formatter to decode message
- Channel sends message through 0 or more server-side sinks
- Terminating sink builds callstack and invokes method on object
- Return value and out params propagate in reverse direction

As you'll see in figure 11.1, which shows both the client-side and object-side of the client-object interaction, the above description details the path the message goes through before it reaches the object. The object developer's job is merely to implement some member function or property that is being invoked. Leading up to this point, the message has been retrieved from the channel, passed through zero or more interceptors (sinks), and delivered to a terminating sink (called a stack builder). This sink then uses the contents of the message to identify which method or property is being accessed, builds an appropriate callstack containing the applicable parameters, then "jumps" into the target method or property implementation.



Note: transparent proxy (TP) has same type hierarchy as target object

Figure 11.1: Method Remoting Overview

AppDomain Boundaries and Objects

Passing object references across AppDomain boundaries requires marshaling

- Formatter consults object's base type and attributes to determine proper handling
- Types that derive from `System.MarshalByRefObject` are AppDomain-bound and result in proxy setup
- Types marked `[Serializable]` are unbound and marshal by value
- No other types may be marshaled - result in `SerializationException` if passed across remoting boundary

Objects, values, and object references are all scoped to a particular AppDomain. When a reference or value needs to be passed to another AppDomain, it must first be marshaled. Much of the CLR's marshaling infrastructure is in the `System.Runtime.Remoting` namespace. In particular, the type `System.Runtime.Remoting.RemotingServices` has two static methods that are fundamental to marshaling: `Marshal` and `Unmarshal`.

When marshaling an object reference, the concrete type of the object determines how marshaling will actually work. As shown in figure 11.2, there are three possible scenarios.

Category	AppDomain-bound	Unbound	Remote-unaware
Applicable Types	Types derived from MarshalByRefObject	Types marked [Serializable]	All other types
Cross-domain marshaling behavior	Marshal-by-reference across AppDomain	Marshal-by-value across AppDomain	Cannot leave AppDomain
Inlining Behavior	Inlining disabled to support proxy access	Inlining enabled	Inlining enabled
Proxy Behavior	Cross-domain proxy Same-domain direct	Never has a proxy	Never has a proxy
Cross-domain identity	Has distributed identity	No distributed identity	No distributed identity

Figure 11.2: Agility and objects

By default, types are remote-unaware and do not support cross-AppDomain marshaling. Attempts to marshal instances of a remote-unaware type will fail.

If a type derives from `System.MarshalByRefObject` either directly or indirectly, then it is AppDomain-bound. Instances of AppDomain-bound types will marshal by reference. This means that the receiver of the marshaled object (reference) will be given a proxy that remotes (forwards) all member access back to the object's home AppDomain. Technically, only access to instance members will be remoted back to the object's home AppDomain. Static methods are never remoted.

Types that do not derive from `MarshalByRefObject` but do support object serialization (indicated via the `[System.Serializable]` pseudo-custom attribute) are considered unbound to any AppDomain. Instances of unbound types will marshal by value. This means that the receiver of the marshaled object (reference) will be given a disconnected clone of the original object. All three behaviors are illustrated in figure 11.3.

```
class A {
  public A() {}
  public Work() {}
}

[ Serializable ]
class B {
  public B() {}
  public Work() {}
}

class C : MarshalByRefObject {
  public C() {}
  public Work() {}
}
```

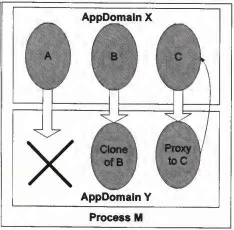


Figure 11.3: Marshaling objects across AppDomains

Channels and formatters

Channels and formatters control how messages get across remoting boundaries and what those messages look like

- Channels can listen for incoming messages and/or send messages out
- Formatters control message layout
- Standard channels are provided (HttpChannel, TcpChannel)
- Standard formatters are provided (SoapFormatter, BinaryFormatter)
- Can build your own channels or formatters (IChannel, IFormatter)

Channel registration

One or more channels must be registered by the client and server before remoting can occur

- Apps must register one or more channels when they start
- Server-side channel(s) start listening for incoming connections
- Objrefs returned to clients contain list of channels available at server
- Channels are registered using `ChannelServices.RegisterChannel` (see figure 11.4)
- `RemotingServices.Marshal` creates a URI suffix-to-object mapping for a given AppDomain

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;

class ClientOrServerApp {
  static void Main() {
    HttpChannel chan = new HttpChannel(999);
    ChannelServices.RegisterChannel(chan);
    ...
  }
}
```

Figure 11.4: Channel Registration

The CLR ships with 2 network channels: `TcpChannel` and `HttpChannel`. These channels can be registered with the runtime either through explicit calls to `ChannelServices.RegisterChannel` or through configuration files. Figure 11.5 shows how to register a channel explicitly as well as how to bind an object to an application-specific identifier. This identifier can then be used as part of the URI that uniquely identifies the object.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels.Http;

class MyApp {
   static void Main() {
    // Instantiate object and bind to "xyzzy"
    TargetType target = new TargetType();
    RemotingServices.Marshal(target, "xyzzy");

    // Register HTTP and TCP channels
    IChannel ch = new HttpChannel(345);
    ChannelServices.RegisterChannel(ch);
    ch = new TcpChannel(3456);
    ChannelServices.RegisterChannel(ch);

    // Object now accessible via http://localhost:345/xyzzy
    // and tcp://localhost:3456/xyzzy
    Console.ReadLine();
  }
}
```

Figure 11.5: Exposing an object via remoting

The proxy architecture of the CLR is fully extensible to support arbitrary communication channels and marshaling formats. The proxy architecture of the

CLR is based on the type `System.Runtime.Remoting.RealProxy`. This type is used to create transparent proxies that turn a method call into two messages: a request message and a response message. Both messages are in-memory abstractions that represent the input and results of the method, respectively. Both message types implement the interface `IMessage` and typically a more specific interface. In the case of cross-AppDomain proxies, these messages are sent over an communications channel that sends the message to the foreign AppDomain. On the AppDomain of the target object, the message is dispatched through a stack builder message sink that unmarshals the contents of the message onto the call stack and then invokes the method. The stack builder sink then turns the results of the call into a message that is then returned to the proxy for unmarshaling to the caller. This architecture is shown in figure 11.6.

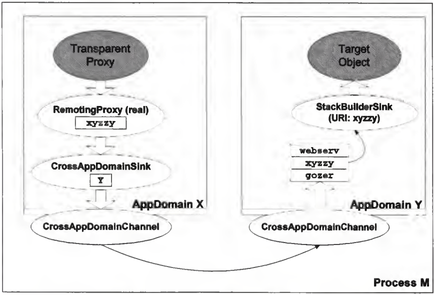


Figure 11.6: Cross-domain method calls

The same-process/cross-AppDomain channel is pre-registered as part of CLR initialization. This same infrastructure can be used to communicate across process and host boundaries by registering network-friendly channels. These channels dispatch messages that come in over the network using the same infrastructure used to dispatch cross-domain messages. This dispatching is shown in figure 11.7.

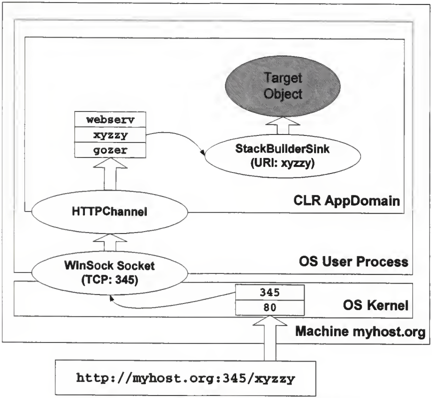


Figure 11.7: Dispatching cross-host method calls

Figure 11.8 shows this URI in use in a client application. Note that assuming the HTTPChannel is used, the application is technically a SOAP-based web service and can be accessed using any platform that supports XML and HTTP. The proxy shown in figure 11.8 can access SOAP-based servers that may or may not be implemented using the CLR.

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

class Client {
  static void Main() {
    ChannelServices.RegisterChannel(new HttpChannel(0));

    // Ask the CLR to build a TP that looks like
TargetType,
    // and that's associated with an RP that knows
    // how to dispatch messages to
http://localhost:345/xyzzy.
    //
    TargetType t = (TargetType)
      RemotingServices.Connect( typeof(TargetType),

"http://localhost:345/xyzzy" );
    try {
      // Call server
      t.DoSomething();
    }
    catch {
      Console.WriteLine("Failed to call server");
    }
  }
}
```

Figure 11.8: Accessing a remote object

Remoting Details

The CLR allows the client- and server-side details of activation and connection activities to be explicitly controlled by the programmer or handled automatically by the runtime by using configuration files.

/enterprise + - compiler switch

Activation options

The CLR supports 3 activation models

- SingleCall Well Known Object - new object per call
- Singleton Well Known Object - same object for all clients
- Client-Activated Object - unique object for each client

With SingleCall well known objects, each incoming message (method call) causes the runtime to instantiate a new object in the server, dispatch the call to that object, then discard that object upon return from the method call. This model is advantageous in a server farm scenario, as it lets each incoming network message be routed to a different machine. Clients specify a type and a URI to locate the object, but no communication is performed until the first method is invoked. During the "connect" call, the runtime just uses the type information to build a transparent proxy to hand back to the client. Only when a method is actually invoked will network communication occur. Because the object is discarded upon return from each method call, the object cannot maintain state on behalf of the client between method calls. And because the runtime controls the instantiation of the object, the client cannot pass constructor arguments into the object.

With Singleton well known objects, the runtime instantiates the server object when the server starts up; prior to any clients connecting. All clients that specify the same URI for a given type will be put into contact with the same object instance in the server. Any state maintained within the object is shared across all clients that connect to the object. And like SingleCall, the client cannot pass constructor arguments into the object.

With Client-Activated types, the object is instantiated as a result of the client-using operator new or `Activator.CreateInstance`. This means that the client can pass constructor arguments into the object. It also means that communication occurs immediately as a result of the activation step. All subsequent method calls made by the client using the resulting proxy will be dispatched to that client's object, which means that the object can maintain state between method calls. When the client releases the proxy, the object will be discarded.

Working with well known objects

Server registers well-known types; clients connect

- Server calls `RegisterWellKnownServiceType` of `RemotingConfiguration` class to publish type-endpoint URI mapping and activation semantics
- Client uses `RemotingServices.Connect` or `Activator.GetObject` to build transparent proxy
- Only the default constructor is supported

Figure 11.9 shows a simple interface (`ICalc`) that the following samples will implement and access.

```
// icalc.dll
//
public interface ICalc
{
  int Add( int a, int b );
}
```

Figure 11.9: The ICalc interface

Figure 11.10 demonstrates the server-side steps to publish a well-known object (with SingleCall activation semantics in this example). In this code, `ChannelServices.RegisterChannel` is used to register a new instance of an HTTP channel object that will be available for connections on port 999. Clients that want to connect to this server will use a connection URL of `http://hostname:999`. Next, `RemotingConfiguration.RegisterWellKnownServiceType` is called to publish the `Calc` type from the `server` assembly, associating this type with the logical endpoint name `calcsrv`. Clients will concatenate this endpoint name on the end of the connection URL they specify to indicate they wish to program against the `ICalc` type. For example, `http://hostname:999/calcsrv`. Since the remoting infrastructure will take care of instantiating our `Calc` class, and handle all of the connection and thread management for us, this server simply blocks in a call to `Console.ReadLine` in order to keep the server process running.

```
// server.exe - references icalc.dll
//
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

class Calc : MarshalByRefObject, ICalc {
  public int Add( int a, int b ) {
    return(a + b);
  }
}

class Server {
  static void Main() {
    ChannelServices.RegisterChannel(new HttpChannel(999));

    RemotingConfiguration.RegisterWellKnownServiceType(
      typeof(Calc), "calcsrv",
      WellKnownObjectMode.SingleCall );

    Console.ReadLine(); // Pressing ENTER exits.
  }
}
```

Figure 11.10: Well Known Object - Server

Figure 11.11 demonstrates the client-side steps to connect to and program against the server. Like the server, the client registers an HTTP channel with the remoting infrastructure. This allows the client to use the HTTP protocol to connect to servers. A zero is passed to the channel constructor to indicate that the channel object can choose any available port. This would only be used in the event that there is a callback relationship between the client and server, where the server will eventually call back to the client. Note that the client in this example uses `RemotingServices.Connect` to specify that a transparent proxy that looks like the `ICalc` type is required. Neither the actual concrete class nor assembly name is required in this example.


```
// client.exe - references icalc.dll
//
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

class Client {
  static void Main() {
    ChannelServices.RegisterChannel(new HttpChannel(0));

    ICalc c = (ICalc)
      RemotingServices.Connect( typeof(ICalc),

"http://localhost:999/calcsrv" );
    try {
      Console.WriteLine("2 + 2 = " + c.Add(2, 2));
    }
    catch {
      Console.WriteLine("Failed to call server");
    }
  }
}
```

Figure 11.11: Well Known Object - Client

Configuration files - well known objects

Registration and connection calls can be delegated to the remoting infrastructure

- Configuration file specifies connection and activation information
- Client and/or server passes name of configuration file to runtime
- Runtime parses file and performs requested registrations
- Facilitates mixing and matching channel, formatter, and message sink types

Figure 11.12 contains a sample client configuration file. Figure 11.13 shows the revised client that uses this configuration file. Figure 11.14 contains a sample server configuration file. Figure 11.15 shows the revised server that uses this configuration file.

```
<!-- client.exe.config -->
<configuration>
  <appSettings>
    <add key="calcURL"
value="http://localhost:999/calcsrv/calcep"/>
  </appSettings>

  <system.runtime.remoting>
    <application name="client">
      <channels>
        <!-- Select the standard HttpChannel that's
specified
             in machine.config.  Could also select "tcp". -
->
        <channel ref="http">
          <clientProviders>
            <!-- Select the BinaryFormatter that's
specified
                 in machine.config.  Could also select
"soap". -->
            <formatter ref="binary" />
          </clientProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Figure 11.12: Remoting configuration file - client

```
// client.exe - references icalc.dll
//
using System;
using System.Configuration;
using System.Runtime.Remoting;

class Client {
  static void Main() {
    RemotingConfiguration.Configure("client.exe.config");

    ICalc c = (ICalc)
      RemotingServices.Connect(
        typeof(ICalc),
        ConfigurationSettings.AppSettings["calcURL"] );

    try {
      Console.WriteLine("2 + 2 = " + c.Add(2, 2));
    }
    catch {
      Console.WriteLine("Failed to call server");
    }
  }
}
```

Figure 11.13: Revised client - using configuration file

```
<!-- server.exe.config -->
<configuration>
  <system.runtime.remoting>
    <application name="calcsrv">
      <service>
        <!-- activation semantics and endpoint for each
type -->
        <wellknown mode="Singleton"
                   type="Calc, server"
                   objectUri="calcep" />
      </service>

      <channels>
        <channel ref="http" port="999">  <!-- HttpChannel
on port 999 -->
          <serverProviders>
            <formatter ref="binary" />   <!-- binary
encoding -->
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Figure 11.14: Remoting configuration file - server

```
// server.exe - references icalc.dll
//
using System;
using System.Runtime.Remoting;

class Calc : MarshalByRefObject, ICalc {
  public int Add( int a, int b ) {
    return(a + b);
  }
}

class Server {
  static void Main() {
    RemotingConfiguration.Configure("server.exe.config");
    Console.ReadLine(); // Pressing ENTER exits.
  }
}
```

Figure 11.15: Revised server - using configuration file

The client and server configuration files allow the server programmer to relocate the details of activation and channel selection out of their code and into a file that can be edited. For the server, the `wellknown` directive provides the parameters that the remoting infrastructure will pass to `RemotingServices.RegisterWellKnownType` on your behalf. Note that the `channel` directive explicitly chooses a port for the server to listen on, whereas the client's configuration file did not specify the port number.

Client-activated types

Client-activated types support dedicated object-per-client usage model

- Client can choose any supported constructor for target type
- Connection established immediately at activation time
- Object can maintain state across method calls
- Server registers client-activated types using `RegisterActivatedServiceType` of `RemotingConfiguration` class
- Client selectively activates type remotely using `Activator.CreateInstance`
- Client may use `RegisterActivatedClientType` to reroute all activations of a given type to a remote location

`RemotingConfiguration.RegisterActivatedServiceType` exports the assembly and type name of a type the server wants to make available via client-activation.

Clients using operator new to connect to and use server objects are not impacted by the server developer changing between well known object semantics and client-activated types. In both cases, the programming model for the client is the same (new, use, discard). However, clients can now pass constructor arguments into the object, since a connection is made immediately when the activation request is made.

Figures 11.16 and 11.17 show a client and server using client-activated types. Figure 11.18 shows a revised client that is leveraging the `RegisterActivatedClientType` method to reroute operator new calls bound to the `Calc` class in the `server` assembly to the remote host and endpoint specified.

```
// server.exe - references icalc.dll
//
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

public class Calc : MarshalByRefObject, ICalc {
  public int Add( int a, int b ) {
    return(a + b);
  }
}

class Server {
  static void Main() {
    ChannelServices.RegisterChannel(new HttpChannel(999));
    RemotingConfiguration.ApplicationName = "calcsrv";

RemotingConfiguration.RegisterActivatedServiceType(typeof(C
alc));
    Console.WriteLine("Server ready.");
    Console.ReadLine();
  }
}
```

Figure 11.16: Supporting client-activated types - server

```
// client.exe - references icalc.dll
//
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Activation;

class Client {
  static void Main() {
    ChannelServices.RegisterChannel(new HttpChannel(0));

    object[] attrs =
      {new UrlAttribute("http://localhost:999/calcsrv")};

    ObjectHandle oh =
      Activator.CreateInstance("server", "Calc", attrs);

    if( oh != null ) {
      ICalc c = (ICalc)oh.Unwrap();
      Console.WriteLine("2 + 2 = " + c.Add(2, 2));
    }
  }
}
```

Figure 11.17: Using a client-activated type without operator new

```
// client.exe - references icalc.dll and server assembly
//
using System;
using System.Runtime.Remoting;

class Client {
  static void Main() {
    RemotingConfiguration.RegisterActivatedClientType(
      typeof(Calc), "http://localhost:999/calcsrv"
    );

    try {
      Calc c = new Calc();
      Console.WriteLine("2 + 2 = " + c.Add(2, 2));
    }
    catch {
      Console.WriteLine("Failed to call server");
    }
  }
}
```

Figure 11.18: CAO client using operator new

Configuration files - client-activated types

Remoting configuration files can also be used with client-activated types

- Server-side configuration lists types that support client activation
- Client-side configuration lists connection and type information

Figure 11.19 contains a sample client configuration file. Figure 11.20 contains a sample server configuration file.

```
<!-- client.exe.config -->
<configuration>
  <system.runtime.remoting>
    <application name="client">

      <!-- Redirect instantiation of Calc type from server
           assembly to remote host and endpoint specified.
-->
      <client url="http://localhost:999/calcsrv">
        <activated type="Calc, server" />
      </client>

      <channels>
        <channel ref="http">             <!-- HttpChannel -
->
          <clientProviders>
            <formatter ref="binary" />   <!-- binary
encoding -->
          </clientProviders>
        </channel>
      </channels>

    </application>
  </system.runtime.remoting>
</configuration>
```

Figure 11.19: Remoting configuration file - client

```
<!-- server.exe.config -->
<configuration>
  <system.runtime.remoting>
    <application name="calcsrv">

      <service>
        <activated type="Calc, server" />
      </service>

      <channels>
        <channel ref="http" port="999">  <!-- HttpChannel
on port 999 -->
          <serverProviders>
            <formatter ref="binary" />   <!-- binary
encoding -->
          </serverProviders>
        </channel>
      </channels>

    </application>
  </system.runtime.remoting>
</configuration>
```

Figure 11.20: Remoting configuration file - server

Note that the `client` and `activated` directives in the client configuration is different for client-activated types than for well-known objects. This directive indicates that use of the `Calc` type from the `server` assembly should be handled using connecting to the `http://localhost:999/calcsrv` endpoint.

The server's configuration file then uses the `service/activated` directives to request the remoting infrastructure to make the `Calc` type from the `server` assembly available as a client-activated type.

Hosting

Remoting infrastructure will not launch surrogate processes automatically

- May write a console application
- May write a WinForms application
- May write an NT service
- NT service provides start-on-boot and process identity configuration
- May leverage IIS and built-in ASP.NET handler
- IIS/ASP.NET provides demand-start configuration and security

ASP.NET Hosting

IIS/ASP.NET can be used to provide "auto-start" support for servers, authentication, and message encryption

- Activation requests routed through IIS to handler registered for `.soap` extension
- Virtual directory mapped onto server directory
- Server directory contains web.config configuration file so handler can find it
- Server assembly must reside in .\bin subdirectory
- WellKnown and Client-Activated types supported
- Limited to HttpChannel
- Any formatter type supported

Using IIS/ASP.NET as the surrogate for your server-side types allows you to make any public class that resides in any assembly (exe or dll) available over the internet, as long as that type derives from MarshalByRefObject and has a constructor that takes no arguments. This means you do not have to write any of the hosting code shown in the Main entry points of each of the sample server applications shown previously. You can simply create a directory on the web server machine, place a properly formatted web.config file (as shown in figure 11.21) in that directory, and place your assembly in the .\bin subdirectory of that directory. Then use the IIS administration tool to create a virtual directory that maps onto the directory where the web.config file is located.

```
<!-- web.config -->
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <!-- Notice the .soap extension for the objectURI.
             This causes ASP.NET to forward the request to
             a built-in HTTP handler. -->
        <wellknown mode="SingleCall"
                   type="Calc, server"
                   objectUri="calc.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Figure 11.21: web.config for use with ASP.NET Hosting

Metadata Availability

Clients and servers must have access to necessary assemblies at runtime

- Interface-only approach useful in W-K scenarios (client only needs interface metadata to build TP)
- Client needs access to/knowledge of concrete types and server assembly when using client-activated types
- Callback/event relationship requires server to have access to client assembly at runtime (hopefully fixed in future release)
- SOAPSUDS.EXE tool can be used to build client-side shim on top of TP

When using WellKnown types, the client only needs build- and run-time access to the assembly containing the interface(s) that are being used to model the client-server interaction. This approach offers the loosest coupling and highest cohesion because the client has no knowledge (either in code or in configuration files) of the server's choice of concrete type or assembly names.

In the current release, the only glitch when using a pure interface-based approach occurs when the client implements an interface that the server knows how to invoke, and passes it's `this` reference as an input argument to a remoted method (via a previously-acquired TP). In this scenario, the client-side serialization of the client's objref will include "too much" information about the client's concrete type name. Subsequently, when the server deserializes such a marshaled objref, the server will encounter a type load failure if the assembly resolver cannot locate the client's assembly on the server machine. Hopefully, this issue will be addressed in a future release.

When using client-activated types, the client needs access to the assembly containing the server's concrete types. The remoting infrastructure will only use the assembly to extract the necessary type information needed to invoke constructors or build TPs - the actual invocation of methods on those types will be remoted as requested.

One alternative to achieving an object-per-client usage model (like client activated types provide), but without requiring the client to either possess apriority knowledge of the server-side concrete type and assembly name, is to use a well-known object as a factory for acquiring an interface onto a new instance of a server-side object. In this scenario, the client would call a method on an interface supported by a well-known server object that causes the server to use operator `new` to instantiate an object and return an interface-based reference to it. Using this approach, the client need only have access to and knowledge of the interface(s) supported by that object, but will still be making calls to a server-side object that has been dedicated to this client's use.

For example, assume the following two interfaces are defined in an assembly that is available to both clients and servers:

```
public interface ICalc
{
    int Add( int a, int b );
}

public interface ICalcFactory
{
    ICalc GetCalc();
}
```

The server might implement these interfaces as follows:

```
class Calc : MarshalByRefObject, ICalc
```

```
{
    public int Add( int a, int b )
    {
        return(a + b);
    }
}

class CalcFactory : MarshalByRefObject, ICalcFactory
{
    public ICalc CreateCalc()
    {
        return new Calc();
    }
}
```

Given the above implementation, the server would only publish the `CalcFactory` type as a well-known object (probably using `Singleton` activation). Any client that then calls `CreateCalc` would receive a proxy to server-side instance of the `Calc` class that has been allocated for their exclusive use. Furthermore, if the `Calc` class had one or more constructors, the `CreateCalc` method would provide equivalent overloads that allow clients to pass those constructor arguments to the server.

This approach provides the same level of functionality of client-activated types, but without the undesirable coupling between client- and server-side code bases.

SOAPSUDS.EXE is a utility that can take a connection string as input, connect to the specified server and endpoint, extract the WSDL (Web Service Description Language) description of the server-side type, and use that information to generate a client-side proxy that acts as a shim over top of the actual transparent proxy. This approach is useful when the client does not have a-priori access to the required server-side types and assemblies as discussed above.

In order for SOAPSUDS to work, the server-side remoting configuration must request that a system-supplied WSDL message sink be installed to handle HTTP requests that include a `?WSDL` query string. Figure 11.22 shows a modified server-side configuration file that registers the WSDL message sink. Figure 11.23 shows the command line for SOAPSUDS.EXE that would connect to the specified endpoint and generate an assembly on the client's machine that could be used to program against the server.


```
<!-- server.exe.config -->
<configuration>
  <system.runtime.remoting>
    <application name="calcsrv">
      <service>
        <!-- activation semantics and endpoint for each
type -->
        <wellknown mode="Singleton"
                   type="Calc, server"
                   objectUri="calcep" />
      </service>

      <channels>
        <channel ref="http" port="999">  <!-- HttpChannel
on port 999 -->
          <serverProviders>
            <formatter ref="soap" />     <!-- SOAP encoding
-->
            <provider ref="wsdl" />      <!-- WSDL support
-->
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Figure 11.22: WSDL-enabled configuration file

```
soapsuds -url:http://localhost/calcsrv/calc.soap?wsdl -
oa:calcproxy.dll
```

Figure 11.23: SOAPSUDS command line

Remoting versus Web Services

Although similar, .NET remoting and Web Services serve different purposes

- Remoting takes a CLR-centric approach to types
- Web Services take an XML schema-centric approach
- Both support HTTP channel
- Remoting also supports TCP and custom channels
- Both support SOAP encoding of messages
- Remoting also supports binary and custom message encoding
- Remoting geared towards CLR <-> CLR distributed systems
- Web Services designed for heterogeneous distributed systems where over-the-wire interop is paramount

11.24

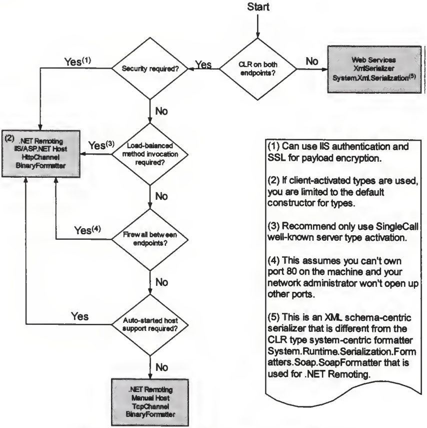


Figure 11.24: .NET Remoting versus Web Services

Summary

- Method remoting is message based
- Singleton and SingleCall activation semantics support for well known objects
- Client-activated types support classic object-per-client model
- IIS/ASP.NET can provide server auto-start support
- Activation details can be specified programmatically or using configuration files
- Remoting useful when both endpoints are CLR-based
- Web Services support heterogeneous interop scenarios

Module 12

I/O and Data Transfer

There are a multitude of ways to transfer data and information into and out of the CLR. At the lowest level is byte-oriented I/O. At the highest level resides XML-based and DBMS-based I/O mechanisms. All of these mechanisms adhere to a common design idiom and usage model.

After completing this module, you should be able to:

- perform binary I/O
- access and update data from a DBMS
- read and write XML-based data
- move from data access to xml-based I/O

Unified I/O

There are a multitude of ways to move information into and out of the CLR. At the lowest level is byte-oriented I/O. At the highest level resides XML-based and DBMS-based I/O mechanisms.

I/O Options

The CLR provides a wide range of options for performing I/O

- `System.IO.Stream`: low-level byte-oriented I/O
- `System.IO.TextReader/Writer`: string-based I/O
- `System.IO.BinaryReader/Writer`: primitive-oriented I/O
- `System.Data.IDataReader`: tabular/rectangular I/O
- `System.Xml.XmlReader/Writer`: semi-structured I/O

The CLR provides a wide range of options for performing I/O, ranging from low-level byte-oriented I/O up through semi-structured XML-oriented I/O. These I/O mechanisms are simply abstract type definitions that are part of the .NET framework. Which mechanism you use often depends on where the information is coming from (or going to). If you are talking to a DBMS, you are likely to use `IDataReader`. If you are reading an unstructured text file from the file system, you are likely to use `TextReader`.

As shown in figure 12.1, the lowest-level interface is `System.IO.Stream`. `System.IO.Stream` exposes a byte-oriented read/write interface that is similar to the Win32 `ReadFile/WriteFile` interface. The `System.IO.BinaryReader/Writer` interface provides an adapter in front of any `System.IO.Stream` object that serializes primitive types and strings into a portable and compact binary format. The `System.IO.TextReader/Writer` interface provides an adapter in front of any `System.IO.Stream` object that performs line-oriented text I/O in a variety of character encoding styles (e.g., UTF8, UTF16).

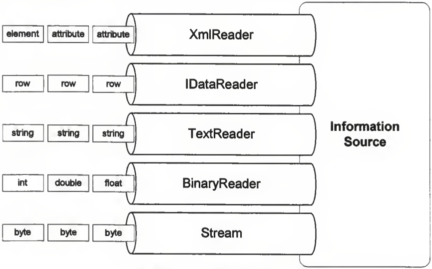


Figure 12.1: Streaming I/O types

`System.Data.IDataReader` does not depend on `System.IO.Stream`. Rather, it presents a table-oriented view over a sequence of uniformly typed rows. Similarly, `System.Xml.XmlReader` also does not depend on `System.IO.Stream`. Instead, it presents a node-oriented view over an XML Infoset. Both `System.Data.IDataReader` and `System.Xml.XmlReader` are discussed in great detail throughout this chapter.

Polymorphism and I/O

I/O is always modeled in terms of abstract types/interfaces to support polymorphism

- Providers of information typically implement abstract type/interface
- Information consumer programs uniformly against abstract type/interface
- Consumer largely ignorant of where information comes from
- Consumer binds to specific provider based on which concrete type is loaded

The I/O interfaces are all defined as abstract data types. This allows a variety of implementations to be plugged into the I/O model without requiring special-case code for each possible data source. As shown in figure 12.2, a consumer is coded against a generic I/O interface, however, that code can operate against a disparate range of data sources. Ultimately, the precise data source is selected at instantiation-time when a concrete implementation class is finally chosen. From that point on, however, the consumer can assume that the underlying implementation conforms to the basic model.

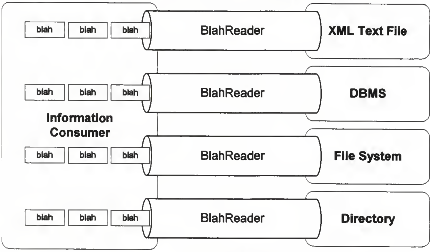


Figure 12.2: Polymorphic I/O

System.Data.IDataReader

Rectangular/tabular I/O is modeled using `System.Data.IDataRecord` and `System.Data.IDataReader`

- `IDataRecord` models a record with named, typed fields/columns
- `IDataRecord` provides type-safe access to fields/columns by name or index
- `IDataReader` adds support for forward-only traversal through records/rows
- `IDataReader` initially positioned prior to first record
- `IDataReader` also supports multiple results (e.g., SQL batches)

The `System.Data.IDataReader` interface is the focal point of the .NET data access stack. `System.Data.IDataReader` is an interface that extends the `System.Data.IDataRecord` interface. These two interfaces combined model type-safe access to tabular data.

Figure 12.3 shows the definitions of `IDataRecord` and `IDataReader`. Note that the `IDataRecord` interface models a particular row or record in a table. Type-safe access to each of the fields/columns is provided. The type and name of each field is also accessible through this interface. `IDataReader` extends `IDataRecord` by adding support for forward-only traversal through a sequence of records/rows. Figure 12.4 shows the basic model of `IDataRecord/IDataReader`.

```
namespace System.Data {
  public interface IDataRecord {
    int FieldCount { get; }
    object this[int colNo] { get; }
    object this[string colName] { get; }

    int GetOrdinal(string colName);
    string GetName(int colNo);
    Type GetFieldType(int colNo);
    string GetDataTypeName(int colNo); // e.g. varchar(32)
    bool IsDBNull(int colNo);
// typed column accessors where T = primitives++
    T GetT(int colNo);
  }
  public interface IDataReader : IDataRecord {
    bool IsClosed { get; }
    int  Depth { get; } // for nested tables
    int  RecordsAffected { get; }

    bool Read();        // advance to next record
    bool NextResult(); // advance to next rowset
    DataTable GetSchemaTable(); // get schema
    void Close();       // release all resources
  }
}
```

Figure 12.3: IDataRecord/IDataReader

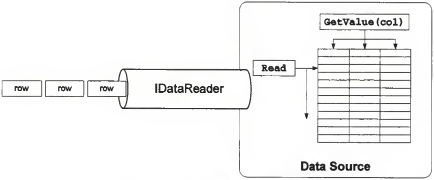


Figure 12.4: IDataReader/IDataRecord

`IDataReader` objects are initially positioned immediately prior to the first record. As shown in figure 12.5, the canonical usage is to write a `while` loop, processing each record until the `Read` method returns false, indicating that there are no more records to consume. As each record is traversed, individual fields are accessed using type-safe accessor methods that accept a zero-based field/column index. To support SQL batch statements that may return multiple tables, `IDataReader` also supports multiple results. For multi-result readers, the reader object is initially positioned at the first table. Subsequent tables are accessible through the `NextResult` property.

```
void PrintTable(System.Data.IDataReader reader) {
  while (reader.Read()) {
    string name = reader.GetString(0);
    double age = reader.GetDouble(1);
    bool dead = reader.GetBoolean(2);
    if (dead)
      Console.Write("The late ");
    Console.WriteLine("{0} is {1} years old", name, age);
  }
}
```

Figure 12.5: Using IDataRecord/IDataReader

System.Data.IDbConnection and friends

`IDataReaders` are typically created by DBMS-style connections and commands

- Connection to data source modeled using `DbConnection` interface
- Executable statement or table name modeled using `DbCommand` interface
- Command object supports parameterized statements using parameter objects
- .NET Framework ships with implementation over raw TDS protocol for SQL Server
- .NET Framework ships with implementation over OLE DB

`IDataReader` objects rarely fall from the sky. Rather, they typically come from data sources that implement the `IDbConnection` family of interfaces. This family of interfaces is modeled loosely on the OLE DB object model. As shown in figure 12.6, each object in the model implements a standard interface as well as optional provider-specific interfaces.

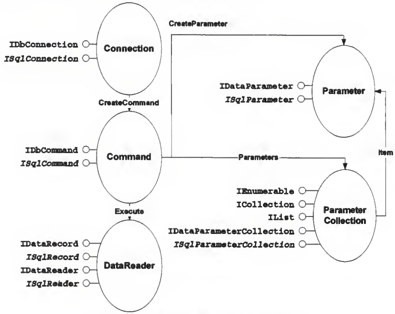


Figure 12.6: IDbConnection and friends

The `IDbConnection` object model begins with a connection object. The connection object models a (secured) connection to a data source. The connection object also acts as the scope for transaction processing. To access data through a connection, one needs a command object. Command objects represent either parameterized SQL statements or simply table/view names. Once a command is associated with a connection, its `Execute` method may be called. This method communicates with the data source and may result in an `IDataReader` object that represents the result of the command (or table).

While multiple implementations of `IDbConnection` are possible, the .NET framework ships with two implementations. One implementation (`System.Data.SqlClient`) communicates directly with SQL Server via SQL Server's proprietary TDS protocol. The other implementation (`System.Data.OleDb`) simply thunks down to OLE DB. Figure 12.7 shows an example of the former provider being used to execute a simple query. Figure 12.8 shows an example of a parameterized update statement.


```
void PrintCustomers() {
  IDbConnection conn = new SqlConnection();
  conn.ConnectionString="data source=localhost;user
id=sa;pwd=;initial catalog=northwind";
  conn.Open();
  try {
    IDbCommand cmd = conn.CreateCommand();
    cmd.CommandText = "select * from customers";
    IDataReader reader = cmd.ExecuteReader();
    try {
      PrintTable(reader); // see earlier fragment
    }
    finally
    { reader.Close();  }
  }
  finally
  { conn.Close(); }
}
```

Figure 12.7: Using IDbConnection and friends

```
void PrintCustomers(IDbCommand cmd, int minAge) {
  cmd.CommandText = "update recs set old=1 where age > @m";
  IDataParameter dbparam = cmd.CreateParameter();
  dbparam.ParameterName = "@m";
  dbparam.DbType = DbType.Int32;
  dbparam.Value = 32;
  cmd.Parameters.Add(dbparam);
  IDataReader reader = cmd.ExecuteReader();
  try {
    PrintTable(reader); // see earlier fragment
  }
  finally
  { reader.Close();  }
}
```

Figure 12.8: Using IDbCommand with parameters

System.Xml.XmlReader

XML-based I/O is modeled using `System.Xml.XmlReader` and `System.Xml.XmlWriter`

- `XmlReader` is pull-mode interface to XML data sources and parsers
- `XmlWriter` is push-mode interface to XML generators and writers
- `XmlTextReader` is the CLR's XML parser
- `XmlTextWriter` eliminates need to generate XML using `WriteLine`
- Other implementations of `XmlReader`/`XmlWriter` are likely

`System.Xml` uses a streaming I/O model. This model assumes that the "provider" of XML (e.g., an XML parser, an XML-based data source) will provide a stream of data items in document order. It also assumes that traversal only happens in one direction: forward. This frees the provider from having to retain unnecessary state once a piece of information has been delivered.

Prior to the .NET framework, the most popular streaming interface suite was the Simple API for XML (SAX). SAX-based providers delivered the stream of data items by invoking methods on an object provided by the consumer. Because all of the method invocations came from the provider, this model is sometimes referred to as a "push-model." Push-model providers require the consumer to keep a state machine across method invocations in order to keep track of "where they are" in the stream of information. For this reason, push-model consumers are often considered harder to design and program. In contrast, pull-model providers keep more of that state machine internally, theoretically making it easier to write a pull-model consumer. A pull-model provider is passive and expects the consumer to make method calls in order to fetch the next piece of information. A pull-model provider typically needs to keep additional cursor state, which in a push-model is retained by the consumer. Pull-model providers are often harder to write but are generally considered easier to use.

`System.Xml` uses two types for performing I/O. `XmlReader` is a pull-model interface that XML producers (including parsers) implement to expose XML-based information according to the Infoset. `XmlWriter` is a push-model interface that XML consumers (including text output streams) implement to consume XML-based information according to the Infoset. The relationship between the two types is shown in figure 12.9.

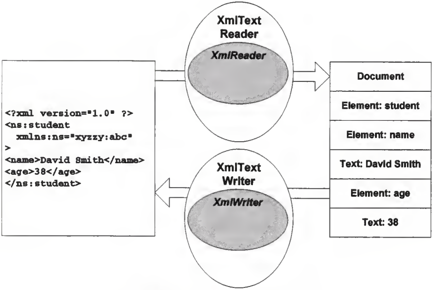


Figure 12.9: Streaming I/O Model

Common wisdom indicates that a pull-model interface pushes complexity into the provider of XML-based information, making it easier for the consumer. Acknowledging this belief, `System.Xml` provides `XmlReader` ,a pull-model interface for exposing XML-based information. More precisely, `XmlReader` is an abstract base class, not an interface, however, the XML stack was designed as if it was an interface (it is an ongoing debate between Microsoft and the community at large as to whether using an abstract base class was the correct decision).

`XmlReader` provides a set of abstract methods and properties for scanning across a stream of XML information items. The `XmlReader` only allows scanning in the forward direction, which keeps the implementation of the producer somewhat manageable. As shown in figure 12.10, the `XmlReader` implementation is expected to keep some notion of the "current" position, and a variety of properties allow the consumer to access various aspects of the "current" information item (e.g., namespace URI, local name, type of item). The primary method of `XmlReader` is the `Read` method, which (like `IEnumerator.MoveNext`, returns `true` as long as the "current" node has not gone off the end of the stream. Figure 12.11 shows an example of a simple program that scans an XML stream and reads the "name" and "age" subelements into a string and double, respectively. Note that this consumer code keeps track of where it is using a single local variable (`eno`) and throws exceptions when the incoming data stream does not match what is expected. The code in

12.12 shows an alternative technique that pushes the verification of element name and position into the underlying `XmlReader` implementation.

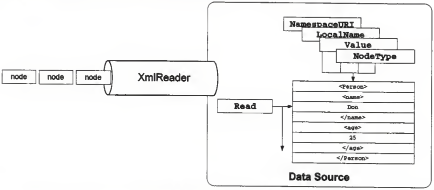


Figure 12.10: XmlReader

```
static void ReadFromReader(XmlReader reader,
                    out String name, out double age)  {
  name = null;  age = 0;  int eno = 0;
  while (reader.Read())  {
    if (reader.NodeType == XmlNodeType.Element) {
      eno++;
      if (eno == 1) {
        if (reader.LocalName != "student")
          throw new XmlException("Parse error");
      }
      else if (eno == 2) {
        if (reader.LocalName != "name")
          throw new XmlException("Parse error");
        name = reader.ReadString();
      }
      else if (eno == 3) {
        if (reader.LocalName != "age")
          throw new XmlException("Parse error");
        age = reader.ReadDouble();
      }
      else
        throw new XmlException("Parse error");
    }
  }
  if (eno != 3)
    throw new XmlException("Parse error");
}
```

Figure 12.11: Using XmlReader

```
  static void ReadFromReader2(XmlReader reader,
                              out String name, out double age)
{
    name = null;
    age = 0;
    reader.ReadStartElement("student", "xyzzy:abc");
    reader.ReadStartElement("name", "");
    name = reader.Value;
    reader.ReadEndElement();
    reader.ReadStartElement("age", "");
    age = reader.Value.ToDouble();
    reader.ReadEndElement();
    reader.ReadEndElement();
  }
```

Figure 12.12: Using XmlReader

`XmlReader` is an abstract type, which means you can program against it but you cannot instantiate objects using `XmlReader` alone. XML providers subclass `XmlReader` and must provide the implementation pieces to make `XmlReader` actually work. There are two "built-in" implementations: `XmlTextReader` and `XmlNodeReader`. `XmlTextReader` uses a user-provided `TextReader` to read an XML 1.0 + Namespaces document and turn it into a stream of information items. `XmlTextReader` is commonly known as an XML parser, as it is the only component in the .NET framework that performs the work classically done by an XML parser. `XmlNodeReader` is used to take a DOM node and expose it as a stream. In general, this doesn't make a lot of sense, since DOM trees are typically big and expensive. However, `XmlNodeReader` allows you to adapt an existing DOM tree into scenarios that expect an `XmlReader`.

Figure 12.13 shows two ways to instantiate an `XmlTextReader`. The first technique layers in the underlying `Stream` and `TextReader` by hand. The second technique relies on the convenience constructor to do the same work on its behalf.

```
static XmlReader OpenXMLReader(String filename) {
  Stream stm = new FileStream(filename, FileMode.Open);
  TextReader tr = new StreamReader(stm);
  XmlReader xr = new XmlTextReader(tr);
  return xr;
}
static XmlReader OpenXMLReader2(String filename) {
  XmlReader xr = new XmlTextReader(filename);
  return xr;
}
```

Figure 12.13: Opening an XmlReader

`XmlWriter` is the push-model version of `XmlReader`. The two types were defined in tandem, and one can easily read information from an `XmlReader` and turn around and write it to an `XmlWriter`. Like `XmlReader`, several concrete implementations exist, and user-defined implementations are inevitable.

In general, it is extremely easy to use `XmlReader`. Figure 12.14 shows a simple program that provides a simple XML document to the underlying `XmlWriter` implementation. If the underlying implementation was an instance of `XmlTextWriter`, then a sequence of angle brackets and tokens would be emitted. If the underlying implementation was an instance of `XmlNodeWriter`, then a DOM tree would be produced. If the underlying implementation were an interface to a DBMS, then perhaps a new record would be inserted into a table.

```
static void WriteToWriter(XmlWriter writer,
                          String name, double age)  {
  writer.WriteStartElement("ns", "student", "xyzzy:abc");
    writer.WriteElementString("name", name);
    writer.WriteElementString("age", age.ToString());
  writer.WriteEndElement();
}
```

Figure 12.14: Using XmlWriter

Figure 12.15 shows two ways to instantiate an `XmlTextWriter`. The first technique layers in the underlying `Stream` and `TextWriter` by hand. The second technique relies on the convenience constructor to do the same work on its behalf.

```
static XmlWriter OpenXMLWriter(String filename) {
  Stream stm = new FileStream(filename,
FileMode.CreateNew);
  TextWriter tw = new StreamWriter(stm);
  XmlWriter xw = new XmlTextWriter(tw);
  return xw;
}
static XmlWriter OpenXMLWriter2(String filename) {
  XmlWriter xw = new XmlTextWriter(filename,
Encoding.Unicode);
  return xw;
}
```

Figure 12.15: Opening an XmlWriter

System.Xml.XPath.XPathNavigator and friends

`System.Xml.XPath.XPathNavigator` is the primary cursor-oriented interface to XML data

- `XPathNavigator` supports backwards traversal through XML stream
- `XPathNavigator` supports filtering based on XPath expressions
- `XPathNavigator` functionality more or less superset of `XmlReader`
- W3C-style DOM still supported largely for legacy work

`XmlNode/XmlDocument` is as close to a W3C DOM as the .NET framework provides. As shown in 12.16, each information item in an XML Infoset has a corresponding type in that extends `XmlNode`. Note that unlike the W3C DOM, which is based exclusively on abstract interfaces, the `XmlNode` hierarchy consists almost exclusively of concrete classes. This means that alternative implementations of `XmlNode` are not really possible. Rather, `XmlNode` and friends are an implementation from Microsoft to keep DOM users happy.

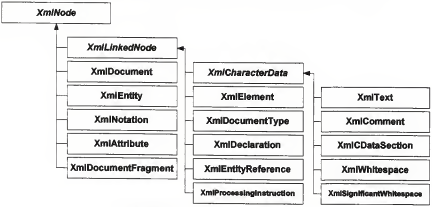


Figure 12.16: XmlNode Hierarchy

Figure 12.17 shows how a `XmlNode` DOM tree can be built from an underlying `XmlReader` implementation. As shown in figure 12.18, one simply needs to call the `XmlDocument.Load` method to translate a stream of XML information into an in-memory DOM tree. Figure 12.19 shows how to use the DOM tree once it has been built.

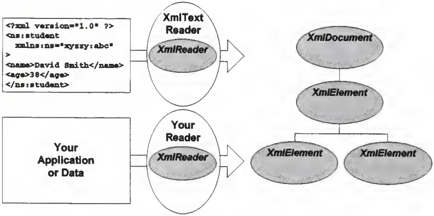


Figure 12.17: XmlDocument.Load

```
static XmlDocument DOMFromReader(XmlReader reader) {
  XmlDocument doc = new XmlDocument();
  doc.Load(reader);
  return doc;
}
```

Figure 12.18: Loading a DOM

```
static void ReadFromDOM(XmlDocument document,
                        out String name, out double age) {
  name = null;   age = 0;
  XmlElement studentElem = document.DocumentElement;
  XmlNodeList children =
studentElem.GetElementsByTagName("*");
  if (studentElem.LocalName != "student"
      || children[0].LocalName != "name"
      || children[1].LocalName != "age"
      || children.Count != 2)
    throw new XmlException("Parse error");

  name = children[0].InnerText;
  age = children[1].InnerText.ToDouble();
}
```

Figure 12.19: Using a DOM

The DOM is largely deprecated in `System.Xml`. Rather, `XmlReader`/`XmlWriter` are the preferred streaming interface suite. To support a richer traversal model, `System.Xml` defines the `XPathNavigator` abstract type. `XPathNavigator` supports a fairly rich set of traversal styles,

including traversal using XPath expressions. `System.Xml` provides `XPathDocument`, which is an implementation of `XPathNavigator` that uses an `XmlReader` as its XML information source. This is shown in figure 12.20.

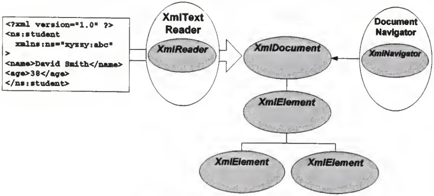


Figure 12.20: XPathNavigator

One of the more useful aspects of `XPathNavigator` is its support for XPath expressions and location paths. Figure 12.21 shows the use of XPath with `XPathNavigator`.

```
static void TouchAllElements(XmlReader reader) {
  IXPathNavigable doc = new XPathDocument(reader);
  XPathNavigator nav = doc.CreateNavigator();
  XPathNodeIterator nodes = nav.Select("//*");
  while (nodes.MoveNext())
    Console.WriteLine("{0} : {1}",
          nodes.Current.NamespaceURI,
          nodes.Current.LocalName);
}
```

Figure 12.21: Using XPathNavigator and XPath

System.Data.DataSet

`System.Data.DataSet` is a disconnected in-memory database

- `DataSet` is a collection of tables and relations
- `DataSet` supports sorting, filtering and arbitrary access
- `DataSet` designed for ISAM/Access die-hards
- Supports XML/XML Schema-based persistence
- Can push changes back into database via provider-specific `IDataAdapter` thunks
- Like the ADO Recordset before it, the `DataSet` is a world unto itself

`System.Data.DataSet` functions as an in-memory database. The `DataSet` has no connection to an underlying data source. Rather, it is meant to be used as a stand-alone database by itself. That stated, it is possible to populate a `DataSet` from an `IDbCommand`-based data provider.

In many ways the `DataSet` is a holdover from the ADO era. Unlike the ADO era, however, the disconnected data model is now very different from the connected model. Given the wide range of options for dealing with in-memory object models and easy metadata-driven persistence, it remains to be seen whether the `DataSet` will be as popular as its predecessor, the ADO Recordset.

Summary

- Bulk/raw I/O modeled using System.IO.Stream abstract type
- Text I/O modeled using System.IO.TextReader/Writer abstract type
- Binary I/O modeled using System.IO.BinaryReader/BinaryWriter concrete type
- Tabular I/O modeled using System.Data.IDataReader abstract type
- Semi-structured I/O modeled using System.Xml.XmlReader abstract type

Module 13

Server-side Programming with ASP.NET

While the initial motivation for the CLR was to address problems in COM, there is no doubt that many developers will rely on ASP.NET as their development platform. ASP.NET provides a non-interpretive environment and framework for web application compilation, execution, and building user interfaces.
After completing this module, you should be able to:

- write HTML and XML-based ASP.NET pages
- configure and deploy ASP.NET-based applications
- use pre-compiled components in an ASP.NET page

ASP.NET Architecture

ASP.NET uses the CLR to replace the existing ISAPI/ASP infrastructure of IIS with a more efficient and easier-to-use framework for servicing HTTP requests. At the same time, ASP.NET provides its own framework for compilation, execution, and building user interfaces.

Evolution

On one hand, ASP.NET is an evolution of the ASP programming model

- Still provides the same intrinsic objects
- Still can mix script and html
- Some ASP code can be ported with no changes
- ASP.NET supports any .NET language
- Script and html is now compiled into an Assembly
- Pages use .aspx extension to distinguish from .asp pages

ASP.NET retains many of the useful features of ASP. It provides the ability to intersperse code and HTML and provides a thin wrapper around the HTTP runtime (Request and Response objects). Even the code that is interspersed within a page is compiled into MSIL, which is then JIT compiled on first access. This provides dramatic speed improvements over interpreted jscript and vbscript, as well as more complete diagnostics because errors are caught at compile time instead of runtime.

- code + html
- msil compiled / jit comp / performance
  errors are caught at comp time

Revolution

ASP.NET is more than just ASP 4.0

- Pages are compiled into assemblies improving performance and diagnostics
- Code-behind encourages better separation of code from HTML
- Extensible, server-side control architecture
- Server-side data binding model
- Form validation architecture
- Web services allow assemblies to expose themselves as SOAP servers

In other ways, ASP.NET provides revolutionary changes to the ASP programming model. For one, it provides a completely new programming model, complete with form support, server side controls, and data binding. Web service support is integrated, allowing Assemblies to expose their functionality as SOAP servers. Every ASP.NET page is an instance of `System.Web.UI.Page`. No more "intrinsics" where variables are magically bound at runtime. All relevant information to a page can be accessed through its base class. For example, `System.Web.UI.Page` has two members called `Response` and `Request`. The ability to integrate .NET code with pages encourages a much cleaner separation of programmatic logic from HTML presentation than ASP did.

ASP.Net page : System.Web.UI.Page

What is ASP.NET?

At a high level, ASP.NET is a collection of .NET classes that collaborate to process an HTTP request and generate an HTTP response

- Some classes are loaded from system assemblies
- Some classes are loaded from GAC assemblies
- Some classes are loaded from local assemblies
- To work with ASP.NET, you must build your own classes that integrate into its existing class structure
- Some of your classes will be in pre-built assemblies
- Some of your classes will be in assemblies generated implicitly from ASP.NET files (aspx, ashx, asmx, ...)

ASP. Net is a collectio
of class. that process
an HTTP request and generate
HTTP response.

To understand what ASP.NET is, and how it works, it is useful to think about it a high level. Fundamentally, ASP.NET is a collection of .NET classes that collaborate to process an HTTP request and generate an HTTP response. Some of these classes are loaded from system assemblies, some from assemblies registered in the Global Assembly Cache (GAC), and some from local assemblies. The way you, as a developer of a web-based application, will work with ASP.NET, is to construct your own classes that integrate into ASP.NET's existing class structure. Some of your classes may reside in pre-built assemblies, either locally or registered in the GAC. Others of your classes will be assemblies that are generated implicitly by ASP.NET from files (like aspx, ashx, or asmx files). It is important to keep this model in mind as you begin to explore ASP.NET features. Figure 13.1 shows a depiction of this high-level view of ASP.NET.

asp. net { pre-built classes
classes generated from aspx pages

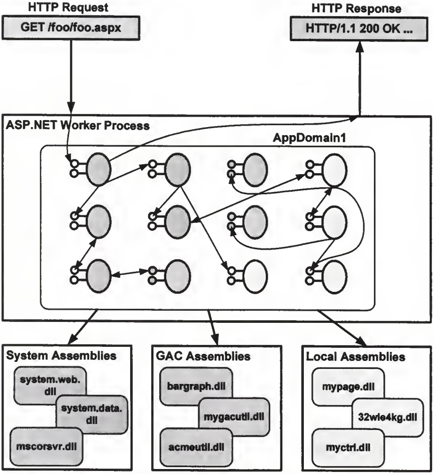


Figure 13.1: High-level view of ASP.NET

ASP.NET basics

Each ASP.NET page is parsed and compiled into a class that extends `System.Web.UI.Page`

- Page class implements `IHttpHandler`
- A lot of the Page class is dedicated to forms/control processing
- Exposes HttpContext properties as own properties

The `System.Web.UI.Page` class is the base class for every page in ASP.NET. It implements the `IHttpHandler` interface to handle incoming requests using the contents of the page. Properties of the `HttpContext` object are exposed indirectly through the page class (like the `Response` and `Request` objects). Figure 13.2 shows a simple ASP.NET page.

```
<%@ page language="C#" %>
<%
// note that this variable refers to subtype of Page
  string s = this.GetType().BaseType.ToString();
// note that Response is a property of Page
  this.Response.Write(s);
%>
```

Figure 13.2: ASP.NET page

IHTTPHandler is implemented by a page class

Page compilation

Every ASP.NET page is compiled into an assembly on first access

- It is compiled into an assembly containing a single class that derives from `System.Web.UI.Page`
- The name for the `Page`-derived class is the file name of the page, replacing the "." with a "_" (like `foo_aspx`)
- Any static HTML along with any interspersed code is rendered in the `Render` function of a single control embedded in the page
- Server-side code blocks in the page become part of the new `Page`-derived class definition.
- The generated assembly is stored in the 'Temporary ASP.NET Files' directory on the server machine

class name = filename_aspx

When a page is requested for the first time, ASP.NET will compile the page into a `System.Web.UI.Page`-derived class contained within an assembly. This assembly will be placed in the CodeGen directory on the server machine (a subdirectory within the system directory) and referenced through a shadowing process that leaves the generated assembly in place so that it can be replaced without shutting down the ASP.NET worker process. This class will be named using the file name of the page that generated it. For example, the file foo.aspx would generate a class called `foo_aspx`. The compilation process takes any static HTML in the page and places it in the `RenderControl()` method of a control within the page. Interspersed code (that is, code added using the <% %> notation) is also added to the `RenderControl()` method of that control. Code that sits within a server-side code block is added to the `Page`-derived class definition.

Figure 13.3 shows the page compilation process from the initial request to the final rendering to the client. When a request first comes in, a static method of the `PageParser` class is called, `GetCompiledPageInstance` to retrieve the `Page`-derived class definition. If the compiled assembly does not yet exist, the page source file is located and the new assembly is generated. If it does exist, it simply uses the existing assembly. Next, an instance of the `Page`-derived class is created and its `ProcessRequest` method is called (inherited from the `IHttpHandler` interface). The invocation of `ProcessRequest` on the page class will ultimately render all of the controls that exist in the page, populating the response object for the client. At this point, the instance of the `Page` class is no longer needed, and is discarded.

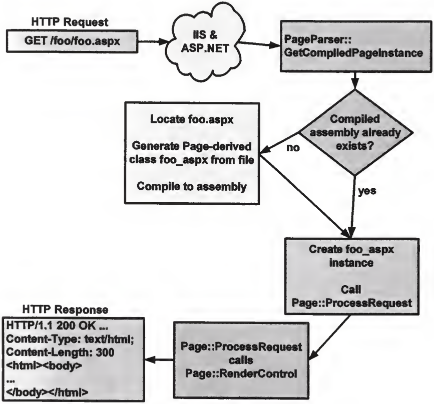


Figure 13.3: ASP.NET Page Compilation

ASP.NET Compilation model

ASP.NET compiles code on demand based on source code dependencies (much like NMAKE)

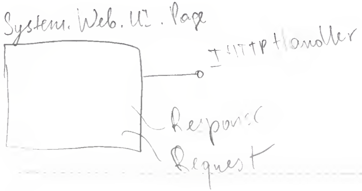
- ASP.NET compiles .ASPX files once and caches the resultant DLL
- If source file(s) newer than cached DLL, a new compilation takes place and the new DLL is cached
- If source file(s) not newer than cached DLL, the cached DLL is used
- Shadow-copies of the DLLs are used to allow existing requests to be processed using the "old" code

Anyone that has built ASP applications has inevitably had to deal with IIS locking any DLLs used by the application. This leads to a frustrating development process as you must shut down and restart IIS each time you want to test a new version of a dll. Similarly, if you need to patch a dll that is already in use on a deployed web application, you must take that application offline to install the dll. If this were the case with ASP.NET, any change to your pages would require you to bounce the ASP.NET worker process as they are all compiled into assemblies. Fortunately, ASP.NET takes care to never load the actual page assembly, rather it uses a sophisticated shadowing technique to load a duplicate copy of any assembly. The shadowing process keeps track of the source file that generated the assembly and will be sure to re-generate the assembly if the source file is updated. Any clients that request that page after the source file has been updated will see the new assembly, while any clients that were already using that page will continue to see the old version of the assembly until they terminate their session and reconnect. This shadowing feature makes updating web applications, even live ones, as simple as copying the new pages and/or assemblies into the appropriate directories on the server machine.

System.Web.UI.Page

The Page class provides facilities for rendering HTML

- `Response` and `Request` objects are available as properties of the class
- Methods for rendering are provided
- Events associated with generating the page are defined



Once you understand that each page you author in ASP.NET will be compiled into a `System.Web.UI.Page`-derivative, you can take advantage of this fact by understanding the features of the `Page` class. Figures 13.4 and 13.5 show some of the more useful methods, properties, and events provided by the `Page` class. Classic ASP programmers will notice that the intrinsic variables that were available to scripts in ASP pages are now defined as properties in the `Page` class. The `Request`, `Response`, `Server`, `Application`, and `Session` properties behave in much the same way that their intrinsic equivalents in ASP behaved. Similar methods like `MapPath` provide the same functionality they do in traditional ASP. Other features of the `Page` class have no direct equivalent in traditional ASP. You can query the capabilities of the client with `ClientTarget`, as well as ask for his credentials using the `User` property. The `IsPostBack` property tells you whether this is the first time a client has requested this page, or if this is a subsequent request generated by the client submitting a Post to the page. Finally, you can define handlers with several events, the most used of which is the `Load` event. The `Load` event is triggered as the page is loaded and before it renders its controls to the `Response` object, giving you a chance to populate controls with data, programmatically generate HTML to the `Response`, or any number of other interesting tasks.

```
class Page : TemplateControl, IHttpHandler
{
        // State management
    public HttpApplicationState Application {get;}
    public HttpSessionState Session {virtual get;}
    public Cache Cache {get;}

        // Intrinsics
    public HttpRequest Request {get;}
    public HttpResponse Response {get;}
    public HttpServerUtility Server {get;}
    public string MapPath(string virtualPath);

        // Client information
    public ClientTarget ClientTarget {get; set;}
    public IPrincipal User {get;}
    //...
}
```

Figure 13.4: System.Web.UI.Page


```
class Page : TemplateControl, IHttpHandler
{
     // Core
    public UserControl LoadControl(string virtualPath);
    public virtual ControlCollection Controls {get;}
    public override string ID { get; set;}

    public bool IsPostBack {get;}
    protected virtual void
               RenderControl(HtmlTextWriter writer);
     // Events
    public event EventHandler Init;
    public event EventHandler Load;
    public event EventHandler PreRender;
    public event EventHandler Unload;

    //...
}
```

Figure 13.5: System.Web.UI.Page

Figure 13.6 shows a sample aspx file that customizes the `Page` class by adding data members and event handlers. Note that code placed within a server-side code block will always be added to the `Page`-derived class (declared variables are turned into member variables, functions are turned into member functions, etc.). Any interspersed code placed using inline script notation will be placed in the `RenderControl` method of the derived `Page` class.

Placed class

Placed in RenderControl method

```
<%@ Page Language="C#" %>
<html><body><head>
<script language="C#" runat=server>
  private ArrayList m_values = new ArrayList();
  void Page_Load(Object sender, EventArgs e) {
    if (!Page.IsPostBack) {
      m_values.Add("v1"); m_values.Add("v2");
      m_values.Add("v3"); m_values.Add("v4");
    }
  }
</script>
</head>
<h2>My test page</h2>
<ul>
<%  for (int i=0; i<m_values.Count; i++)
      Response.Write("<li>" + m_values[i] + "</li>");
%>
</ul>
</body> </html>
```

Figure 13.6: Sample aspx file customizing Page

Code behind

In addition to customizing the generated Page class using embedded code, ASP.NET supports page inheritance

- Technique of `Page` inheritance is called code-behind
- Supported through the `Inherits` attribute of the `Page` directive
- Promotes separation of code and presentation
- Code-behind files can either be pre-compiled, or compiled on demand using the src attribute of the `Page` directive

Now that you know that the code you add to an aspx file is going to be compiled into a `Page`-derived class, the next logical step is to actually author the `Page`-derived class yourself, instead of having it done implicitly for you. ASP.NET supports (and encourages) this model through a technique called code behind. Instead of adding methods and event handlers within server-side code blocks in the page file, you define a new class that derives from `System.Web.UI.Page` and define the methods, data members, and event handlers directly in your class. To use this class, the `@Page` directive supports the `Inherits` attribute with which you can specify your `Page`-derived class as the base class for this page. Using this technique, you introduce another class between the `System.Web.UI.Page` class and the class that will be generated by ASP.NET when the aspx file is accessed.

Figures 13.7 and 13.8 show a complete code behind implementation of the aspx file shown earlier. The promise of using code behind to build your pages is that you can separate the logic of a page from its presentation, making it simpler for designers to modify the layout and appearance of a page without having to worry that they are accidentally changing some of the embedded code. It also makes for much more readable pages by removing much of the interspersed code from the HTML controls.

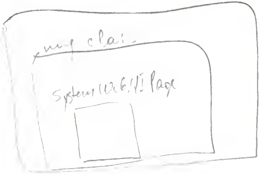
```
<%@ Page Language="C#" src="SamplePage.cs"
    Inherits="DM.AspDotNet.MyPage" %>

<html>
<body>
<h2>My test page</h2>

<% WriteArray(); %>

</body>
</html>
```

Figure 13.7: Sample aspx file with code behind



```
using System;
using System.Web.UI;
using System.Collections;
namespace DM.AspDotNet
{
public class MyPage : Page {
  private ArrayList m_values = new ArrayList();
  protected void Page_Load(Object sender, EventArgs e)  {
  if (!Page.IsPostBack) {
      m_values.Add("v1"); m_values.Add("v2");
      m_values.Add("v3"); m_values.Add("v4");
    }
  }
  public void WriteArray() {
    Response.Write("<ul>");
    for (int i=0; i<m_values.Count; i++)
      Response.Write("<li>" + m_values[i] + "</li>");
    Response.Write("</ul>");
  }
} }
```

Figure 13.8: Sample code-behind file - SamplePage.cs

ASP.NET directives

ASP.NET supports a variety of directives to control compilation and linkage

- All assemblies in the `bin` subdirectory are automatically referenced
- `csc.exe` /r command-line parameter exposed via `@Assembly` directive
- C# `using` statement exposed via `@Import` directive
- Several techniques available for referencing secondary source files

In many ways, ASP.NET takes over the build process, most of which may run post-deployment time. To address this, ASP.NET provides a family of directives that allow the developer to control various aspects of the build process. These directives are listed in figure 13.9. Note that the `@import` directive replaces the `using` statement from C#. Also, the `@assembly` directive replaces the `/r:` command-line switch from `csc.exe` and `vbc.exe`. The `@assembly` directive is not needed for assemblies that reside in the `bin` subdirectory.

Directive Name	Description
`@Page`	Compilation and processing options
`@Import`	Replaces C#'s `using` statement
`@Assembly`	Replaces `csc.exe`'s `/r:` switch
`@OutputCache`	Controls ASP.NET's output caching
`@Register`	Registers tag prefix/name to class name

```
csc.exe /r:bob.dll /r:steve.dll main.cs aux.cs
```

```
using DevelopMentor;
using DevelopMentor.Web;
// code goes here
```

```
<%@page language="C#" %>
<%@import namespace="DevelopMentor" %>
<%@import namespace="DevelopMentor.Web" %>
<%@assembly name="bob" %>
<%@assembly name="steve" %>
<%@assembly src="aux.cs" %>
// code/content goes here
```

Figure 13.9: ASP.NET Directives

One of the more often used directives is the `@page` directive, whose attributes are shown in figure 13.10. The `@page` directive controls programming language, auxiliary source file, MIME content type, and a variety of other parameters.

Name	Description
`language`	Programming language to use for <%
`buffer`	Response buffering
`contenttype`	Default Content-Type header (MIME)
`enablesessionstate`	Turns session state on/off
`transaction`	Controls COM+ transaction affiliation
`src`	Indicates source file for code-behind
`inherits`	Indicates base class other than Page
`errorpage`	URL for unhandled exception page

```
<%@page language="C#" buffer="true" contenttype="text/xml" %>
```

```
<%@page language="C#" src="mycode.cs" inherits="MyBasePage" %>
```

Figure 13.10: @page Directives

Summary

- ASP.NET is a development platform unto itself
- ASP.NET has its own conventions, build process, I/O, and user interface framework
- ASP.NET is likely to be the dominant platform for C#/VB development

Module 14

The HTTP Pipeline

ASP.NET uses the CLR to replace the existing ISAPI/ASP infrastructure of IIS with a more efficient and easier-to-use framework for servicing HTTP requests. After completing this module, you should be able to:

- use the ASP.NET object model to produce a text-based response
- understand the role of HTTP handlers
- understand the role of HTTP modules

HTTP Pipeline

ASP.NET is built on a core set of classes and interfaces that abstract the HTTP protocol. The three core abstractions are the context (HttpContext) that represents the current HTTP request, handlers (classes that implement IHttpHandler) that are capable of servicing HTTP requests, and modules (classes that implement IHttpModule) that can pre/post-process HTTP requests to provide additional services.

HTTPContext represents current request
IHTTPHandler service HTTP requests
IHTTPModule pre/post process HTTP requests

ASP.NET and IIS

ASP.NET uses the CLR to replace IIS's ISAPI/ASP architecture

- User-defined handler objects used to dispatch HTTP requests
- Requests dispatched through ASP.NET-provided ISAPI extension (`aspnet_eisapi.dll`) - one per cpu
- Handlers run in an ASP.NET-provided worker process (`aspnet_ewp.exe`)
- Many IIS features bypassed in favor of ASP.NET-provided features (WAM-based process isolation, ASP object model, and session management)

aspnet_eisapi.dll - extension

aspnet-eisapi.dll - extension

aspnet_ewp.exe - process where handlers run

HTTPContext
HTTPHandler ≈ extension
HTTPModule ≈ filter
HTTPRequest/HTTPResponse

Many IIS features are bypassed in favor of ASP.NET-provided features (in fact, IIS is not technically required to make ASP.NET work). IIS features such as WAM-based process isolation, ASP object model and script hosting, and ASP-managed session state are abandoned in favor of new implementations of the same ideas. That stated, IIS is currently used to dispatch incoming requests through the ASP.NET runtime. In particular, all ASP.NET-based requests are dispatched through an ASP.NET-provided ISAPI extension (`aspnet_eisapi.dll`). This ISAPI extension then forwards the request to an ASP.NET-managed worker process (`aspnet_ewp.exe`), as shown in figure 14.1. In order to process requests, ASP.NET sets up a number of classes to represent various aspects of the request and response. Figure 14.2 shows the various classes that are created and how they relate to each other.

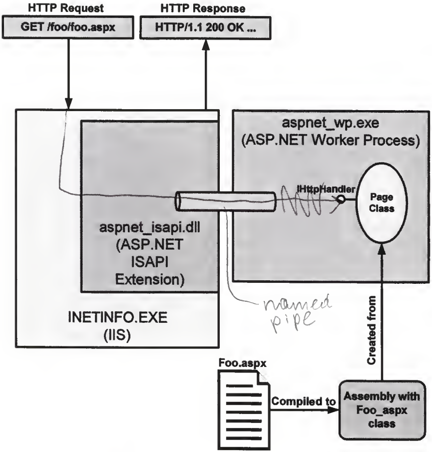
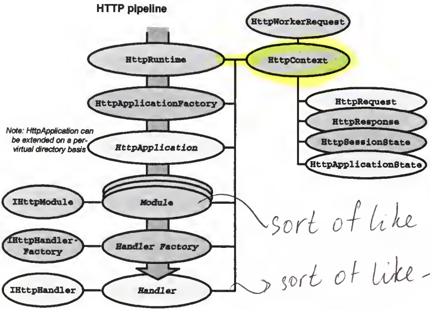


Figure 14.1: ASP.NET Architecture

HTTP handlers are hosted in aspnet_wp.exe
named pipe between iis and aspnet_wp.exe



sort of like filter

sort of like ext.

Figure 14.2: Classes in the HTTP Pipeline of ASP.NET

Context: Request
Response
Session State
Application State

HttpHandlers

ASP.NET uses handler objects to dispatch HTTP requests to user-provided code

- URI paths may be bound to classes that implement the `System.Web.IHttpHandler` interface
- URI paths not bound to a class will be compiled to produce handlers if possible (as happens with aspx files)
- For handlers to be invoked, URI path must be first mapped to the ASP.NET ISAPI Dll (aspnet_isapi.dll)
- Handlers must be listed in the `httpHandlers` section of web.config as well

ASP.NET uses handler objects to dispatch HTTP requests to user-provided code. You may bind a URI path to a class that will handle the incoming request. That class is expected to support `System.Web.IHttpHandler`, which provides the ASP.NET infrastructure with a way of dispatching the incoming request to your code. If the incoming request URI is not mapped to a specific handler class, then the ASP.NET default handler will treat the target URI as a filename and compile it on the fly to produce a handler based on the content and code found in that file.

For each request that comes in, if the requested URI maps to an ASP.NET extension, ASP.NET will look for an IHttpHandler implementation in the web.config file associated with that extension. Figure 14.3 shows a sample web.config file that directs any requests for files with the .foo extension to be handled by the `DM.AspDotNet.FooHandler` class in the assembly "FooHandler.dll". This assembly must be stored in the bin directory of the application (or registered in the GAC) for this technique to work. Also, the .foo extension must be associated with the ASP.NET ISAPI dll (aspnet_isapi.dll) to bootstrap the process.

```
<configuration>
  <system.web>
    <httpHandlers>
       <add verb="*" path="*.foo"
            type="DM.AspDotNet.FooHandler, FooHandler" />
    </httpHandlers>
  </system.web>
</configuration>
```

Figure 14.3: Web.config file defining custom HttpHandler class

Figure 14.4 shows the IHttpHandler interface, as well as the `IHttpHandlerFactory` interface. Figure 14.5 shows a sample implementation of IHttpHandler that displays a simple message when a response is requested. If a client makes a request for a file with the "foo" extension for this application, this handler would be created by the generic HandlerFactory and its `ProcessRequest` method would be invoked.

```
namespace System.Web {
  public interface IHttpHandler {
    void ProcessRequest(HttpContext ctx);
    bool IsReusable {get;}
  }
  public interface IHttpHandlerFactory {
    IHttpHandler GetHandler(HttpContext context,
                            string requestType,
              string url,
                 string pathTranslated);
    void ReleaseHandler(IHttpHandler handler);
  }
}
```

Figure 14.4: IHttpHandler/IHttpHandlerFactory

```
using System;
using System.Web;

namespace DM.AspDotNet
{
  public class FooHandler : IHttpHandler
  {
     public void ProcessRequest(HttpContext context)
     {
       context.Response.Write("My cust handler!");
     }

     public bool IsReusable
     {
       get { return true; }
     }
  }
}
```

Figure 14.5: Sample IHttpHandler implementation

can use
ashx instead
of

HttpContext

ASP.NET object model based on `System.Web.HttpContext`

- `HttpContext` is a superset of the classic ASP object model
- One `HttpContext` object exists for each request being serviced
- The correct `HttpContext` object is passed explicitly to HTTP handlers
- The current `HttpContext` object is exposed via the static property `HttpContext.Current`
- Everything you need to know about the request (and the surrounding application) is exposed via the `HttpContext` object

HTTPContext <-> 1 per each request

Context is passed to HTTP handlers

HTTPContext. Context is exposed via a static property

Incoming calls are associated with a context object for the life of the call. This context object is an instance of the class `System.Web.HttpContext`. The context object maintains information about the incoming request and allows the developer to generate a response programmatically. This is accomplished by providing your own handler object that the context uses to service the call. This handler object can either be directly instantiated by the ASP.NET infrastructure or by a custom factory object that you provide.

The key to understanding request processing in ASP.NET is to focus on the context object. As shown in figure 14.6, the context object has a fair number of properties, each of which exposes some unique facet of the HTTP request being processed. The `Application`, `Session`, `Request`, `Response`, and `Server` properties correspond (roughly) to their classic ASP counterparts. Figure 14.7 shows a demonstration of context usage.

Type	Name	Description
HttpContext	Current (static)	Context for request currently in progress
HttpApplicationState	Application	Appliction-wide property-bag
HttpApplication	ApplicationInstance	Application context (modules)
HttpSessionState	Session	Per-client session state
HttpRequest	Request	The HTTP request
HttpResponse	Response	The HTTP response
IPrincipal	User	The security ID of the caller
IHttpHandler	Handler	The handler dispatched to the call
IDictionary	Items	Per-request property-bag
HttpServerUtility	Server	URL Cracking/Sub-handlers
Exception	Error	The 1st error to occur during processing

Figure 14.6: HttpContext Properties

```
public class MyClass
{
  void SomeFunction()
  {
    HttpContext thiscall = HttpContext.Current;
    thiscall.Response.Write("Hello, ");
    string name = thiscall.Request.QueryString["name"];
    thiscall.Response.Output.WriteLine(name);
  }
}
```

Figure 14.7: HttpContext Example

The most novel property is the `ApplicationInstance` property. This property exposes the application-wide object model, which consists primarily of HTTP modules. An HTTP module is a "global object" that was configured as part of the application using a `web.config` file. Each module is an object that implements the `IHttpModule` interface. HTTP modules serve the same role as ISAPI filters do in classic IIS. Modules are given the opportunity to intercept each step of the request processing, much as an ISAPI filter is.

HttpModules

HttpModules provide pre/post processing on every request

- Exist at the application level (not per request)
- Analogous to ISAPI Filters
- Implement IHttpModule interface
- `Init` method called on application startup allowing module to hook application-level events
- System provided modules include `CookielessSessionModule`, `UrlAuthorizationModule`, and so on

Figure 14.8 shows the relationship between the applications, modules, and handlers. HttpModules exist at the application level, not per request, and are used to preprocess (and/or postprocess) requests before they are sent to handlers. This model is analogous to the role that ISAPI filters play with IIS. HttpModules must implement the IHttpModule interface, shown in figure 14.9. The system in ASP.NET uses modules to implement several top-level features, as shown in figure 14.10. All of these features are provided prior to handlers (or pages) being invoked.

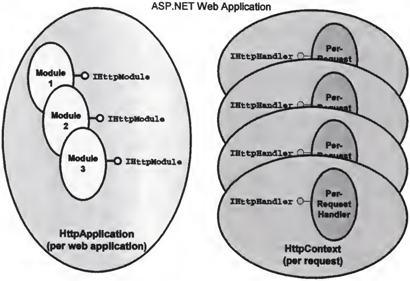


Figure 14.8: Web Application Object Model

```
public interface IHttpModule
{
  void Dispose();
  void Init(HttpApplication context);
}
```

Figure 14.9: The IHttpModule interface

```
<httpModules>
 <add name="OutputCache"
  type="System.Web.Caching.OutputCacheModule, ..." />
 <add name="Session"
  type="System.Web.SessionState.SessionStateModule..." />
 <add name="WindowsAuthentication"

type="System.Web.Security.WindowsAuthenticationModule.."/>
 <add name="FormsAuthentication"
  type="System.Web.Security.FormsAuthenticationModule..."
/>
 <add name="PassportAuthentication"

type="System.Web.Security.PassportAuthenticationModule.."/>
 <add name="UrlAuthorization"
  type="System.Web.Security.UrlAuthorizationModule..." />
 <add name="FileAuthorization"
  type="System.Web.Security.FileAuthorizationModule..." />
</httpModules>
```

Figure 14.10: System-provided modules

Implementing an HTTP module

HTTP modules process events about HTTP request and response message progress through pipeline

- Modules are simply classes that implement `IHttpModule`
- `Init/Dispose` called to connect/disconnect event handlers from `HttpApplication` object
- Specific event handlers added as needed

HTTP modules process events about the progress of HTTP request and response messages through the pipeline. An HTTP module is simply a class that implements the `System.Web.IHttpModule` interface, as shown in Figure 14.11. The `Init` and `Dispose` methods are called when a module is attached to and detached from an `HttpApplication` object, respectively. These methods give the module an opportunity to register and unregister handlers for one or more of the `HttpApplication` object's events. In this example, the `MyModule` class registers for the `BeginRequest` and `EndRequest` events. The handlers for both these events, `OnBeginRequest` and `OnEndRequest`, take two arguments, a reference to the firing object and a reference to a `System.EventArgs` object. The former refers to the module's `HttpApplication` object. The latter is an empty instance of the `EventArgs` class.

```
public class MyModule : IHttpModule
{
  // IHttpModule members
  public void Init(HttpApplication httpApp)
  {
    httpApp.BeginRequest +=
        new EventHandler(this.OnBeginRequest);
  }
  public void Dispose() {}

  // event handlers
  public void OnBeginRequest(object o, EventArgs ea)
  {
    HttpApplication httpApp = (HttpApplication) o;
    HttpContext ctx = (HttpContext) ea.ExtendedInfo;   httpApp.Context.
    ... // do something interesting
  }
}
```

Figure 14.11: Implementing an HTTP module

Deploying an HTTP module

Once an HTTP module is implemented, it must be deployed

- Module's assembly must be put in target virtual directory's `bin` subdirectory or GAC
- `<httpModules>` element must be added to virtual directory's `web.config` file

Once an HTTP module has been implemented, it must be deployed. This involves two steps.

First, the assembly that contains the module must be put in the bin subdirectory of the virtual directory through which it will be accessed or in the Global Assembly Cache (GAC).

Second, an `<httpModules>` element must be added to the virtual directory's `web.config` file, as shown in Figure 14.12. In this example, the `<httpModules>` element contains a single `<add>` element. The `<add>` element has a `name` attribute, which identifies the module in an instance of the HTTP pipeline, and a `type` attribute that provides the fully scoped name of an HTTP module class, and, optionally, the name of the assembly that implements it.

```
<configuration>
  <system.web>
    <httpModules>
      <add name="myModule"
            type="MyNamespace.MyModule, MyAssembly" />
    </httpModules>
     </system.web>
</configuration>
```

Figure 14.12: Configuring HttpModules

Terminating request handling

HTTP module can stop flow of request through pipeline by calling `HttpApplication.CompleteRequest`

- Processing stops after current event completes
- Does not stop other modules from receiving current event
- `EndRequest` event and following events still fire

In some situations, an HTTP module may want to stop the flow of a request through the pipeline. For instance, a security module might want to stop an HTTP request message if the client who is sending it has not been authorized. If an HTTP module wants to short-circuit the processing of an HTTP request, it can call the `HttpApplication.CompleteRequest` method. This call tells the `HttpApplication` object to stop processing the request immediately after the current event is done being processed, e.g., after the `BeginRequest` event completes. Figure 14.13 provides an example.

```
public class SOAPModule : IHttpModule
{
  // IHttpModule members
  public string ModuleName { ... }
  public void Init(HttpApplication httpApp)
  {
    httpApp.BeginRequest +=
        new EventHandler(this.OnBeginRequest);
  }
  public void Dispose() { ... }

  public void OnBeginRequest(object o, EventArgs ea)
  {
    HttpApplication httpApp = (HttpApplication) o;
    HttpContext ctx = (HttpContext) ea.ExtendedInfo;
    if (ctx.Request.Headers["SOAPAction"] == null)
    {
      httpApp.CompleteRequest();
      ctx.Response.StatusCode = 500;
      ctx.Response.StatusDescription =
                        "Internal Server Error";
    }
  }
}
```

Figure 14.13: Calling CompleteRequest

There are a couple of important observations to make about `CompleteRequest`. First, if an HTTP module calls `CompleteRequest`, it should also emit a valid HTTP response message. Second, if an HTTP module calls `CompleteRequest` in a given event handler (its `BeginRequest` handler for instance), any other modules registered to as handlers for that event still receive their notification because the `HttpApplication` class fires events using `MulticastDelegate` objects. Third, even when an HTTP module diverts normal request processing, the `HttpApplication` class still fires the `EndRequest`, `PreSendRequestHeaders` and `PreSendRequestContent` events.

Summary

- ASP.NET uses the CLR to host code that responds to HTTP requests
- ASP.NET object model based on `System.Web.HttpContext`
- ASP.NET uses handler objects to dispatch HTTP requests to user-provided code
- HttpModules provide pre/post processing on every request

Module 15

WebForms

This module looks at the overall architecture of ASP.NET, We will look at the page compilation process, code behind, page rendering, and the ASP.NET object model. We will also look at techniques for building your own custom controls in ASP.NET: how to construct custom binary controls, composite controls, and how to author user controls.

After completing this module, you should be able to:

- understand the ASP.NET object model and where pages fit in that model
- use code behind to separate out web page logic from presentation
- use directives to control the compilation of pages
- understand how controls fit into the ASP.NET architecture
- author custom controls by deriving classes from the Control class
- use custom controls from any ASP.NET page
- author custom composite controls in either code or through an .ascx page as a user control

Server-Side Controls

Web application development has always involved extracting values from query strings and form data. ASP.NET provides a more traditional application development model through server-side controls.

Traditional HTML generation

Traditional ASP development involves explicitly generating client-side HTML

- Server-side script writes directly to intrinsic `Response` object
- Database queries must be turned into HTML elements
- Form contents must be explicitly queried through `Request` object
- Provided little more than server-side `printf` functionality

ASP allows you to intersperse server-side code with HTML, generating portions of a page dynamically. Each time a client submits a request to your asp page, you have an opportunity to generate the appropriate HTML response. Unfortunately, this usually ends up resulting in hard-to-decipher pages with confusing, spaghetti code. Figure 15.1 shows an example of an asp page written to process a user's selection and print a response. Note that the generation of HTML is very explicit in this example. For example, in order to restore that selection box to what the user selected, it is necessary to write the "selected" attribute to only one of the selection items. This is done by conditionally testing the request object's query string to find out what the user selected, and writing the word "selected" to the appropriate selection within the `option` tag. This figure also demonstrates how to conditionally generate HTML based on whether the user has actually filled in the form yet or not. In this case, we want to greet the user with a simple string once he has filled in the form elements. To do this, we test one of the query string values (in this case, the "Name" element) to verify that the form has been submitted. If it has, we generate the string to greet the user. In many ways, the ASP model is little more than a sophisticated `printf` statement, letting you generate customized HTML based on elements in the `Request` object.

```
<%@ Language="javascript" %>
<html><body><form>
    <h3>Enter name: <input name="Name" type=text
        value="<%=Request("Name") %>"></input>
    Personality: <select name="Personality">
        <option
            <% if (Request("Personality") == "extraverted")
{
            Response.Write("selected");
            }%> >extraverted</option>
        <option
            <% if (Request("Personality") == "introverted")
{
                Response.Write("selected");
            }%> >introverted</option>
        <option
            <% if (Request("Personality") == "in-between")
{
                Response.Write("selected");
            }%> >in-between</option>

      </select>
    <input type=submit name="Lookup"
value="Lookup"></input>
    <p><% if (Request("Name") != "") { %>
        Hi <%=Request("Name") %>,
           you selected <%=Request("Personality") %>
      <% } %></p>
  </form>
</body></html>
```

Figure 15.1: Traditional ASP page

ASP.NET server-side controls

ASP.NET defines server-side controls to abstract HTML generation process

- Server-side controls retain state between post-backs
- Server-side controls issue events on the server
- Server-side controls generate appropriate HTML to client

ASP.NET brings a more traditional style of form development to web programming. Instead of asking the developer to manually extract form elements and re-populate controls, ASP.NET defines the concept of server-side controls. Server-side controls retain their state between post-backs so that explicitly accessing the query string and re-generating HTML is no longer necessary. Figure 15.2 show the same page written earlier in ASP re-written in ASP.NET using server-side controls. Programming a page using server-side controls is much more like traditional programming, where the retention of state is assumed. Note that within the server-side code block, we are able to access controls and their values, just as you might on a traditional form-based application. Server-side controls are even capable of generating events on the server, in response to user interaction.

```
<%@ Page Language="C#" %>
<html>

<body>
  <form runat=server>
    <h3>Enter name:
        <input ID="txtName" type=text runat=server/>
    Personality: <select ID="Personality" runat=server>
                 <option>extraverted</option>
                 <option>introverted</option>
                 <option>in-between</option>
                 </select>
        <input type=submit value="Submit"/>
    <p>
      <% if (IsPostBack) {%>
        Hi <%=txtName.Value%>, you selected
           <%=Personality.Value%>
      <% } %>
    </p>
  </form>
</body>
</html>
```

Figure 15.2: ASP.NET page using server-side controls

How is state retained in server controls? Figure 15.3 shows the actual HTML that is presented to the client browser. Note that the form was given a distinct name ("ctrl1"), assigned a default method of 'post' (where a traditional ASP page assumes a "get"), and set the action of the form to the page itself. Even more interesting, is the hidden field named "__VIEWSTATE" which appears with some unreadable value. Many of the values in a form will be posted back to the server as part of the post back process, but some elements, like labels, selections in a list, or display attributes of controls are not posted back. This

hidden field is used to store the state of those controls that do not have their values posted back to the server.

```
<html>
<body>
  <form name="ctrl1" method="post"
        action="ServerSideCtrls.aspx" id="ctrl1">
<input type="hidden" name="__VIEWSTATE"
       value="YTB6MTY0MDA4NTc2Ml9fX3g=a7d02a14" />
    <h3>Enter name: <input name="txtName" id="txtName"
                         type="text" value="Joe" />
    Personality: <select name="Personality"
                         id="Personality">
    <option value="extraverted">extraverted</option>
    <option selected value="introverted">introverted
    </option>
    <option value="in-between">in-between</option>
</select>
    <input type=submit value="Submit"/>
    <p>
        Hi Joe, you selected introverted
    </p>
  </form>
</body>
</html>
```

Figure 15.3: HTML generated by server-side controls

Server-side events

In addition to retaining state, server-side controls can issue events

- Many events traditionally generated on the client-side can be propagated to the server
- Primarily "click" and "change" events are available
- Events that are not "auto-postback" queue up on the client until a postback occurs
- Warning: every server-side event generates a post-back round trip

how are values posted back?

To complete the model of a traditional form-based application, ASP.NET also supports server-side events. This means that you can author handlers for events in server-side script blocks to be invoked when those events occur. Figure 15.4 shows an example of trapping the `ServerClick` event of an input button running on the server. Server events are indicated within HTML tags by prefixing the event name with "On." Every event in ASP.NET uses the same signature, receiving an Object (the entity that sent the event) and a list of event arguments, passed through the `EventArgs` parameter. Most server events use the generic `EventArgs` base class, which is basically a placeholder and contains no additional event information. Those controls issuing events with additional information will pass a derivative of `EventArgs` containing additional event information.

```
<%@ Page Language="C#" %>
<html>
<head>
<script runat=server>
  void WriteGreeting(Object sender, EventArgs E)
  {
    Greeting.InnerText = "Hi " + txtName.Value +
                  ", you selected " + Personality.Value;
  }
</script>
</head>
<body>
  <form runat=server>
    <h3>Enter name: <input ID="txtName"
                           type=text runat=server/>
    Personality: <select ID="Personality" runat=server>
                 <option>extraverted</option>
                 <option>introverted</option>
                 <option>in-between</option>
                 </select>
    <input type=button value="Submit"
            runat=server OnServerClick="WriteGreeting" />
     <div id="Greeting" runat=server />
  </form>
</body>
</html>
```

Figure 15.4: Using server-side events

While this model is an appealing one, especially to developers accustomed to building desktop form applications, it is critical to bear in mind that every server event issued will generate a post-back round trip from the client. Server events should be used judiciously, typically only where in the past you may have performed a manual form post. If you look at the resulting HTML

generated by this ASP.NET page in figure 15.5 you will notice that an explicit `submit()` is issued whenever this button is pressed. Also present are two additional hidden fields for storing the event target and arguments. These are used by controls that need to propagate additional information during an event, and to issue multiple events on a single postback if non "auto-postback" events have been issued as well.

```
<html><body><form name="ctrl2" method="post"
        action="ServerSideEvents.aspx" id="ctrl2">
<input type="hidden" name="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" value="" />
<input type="hidden" name="__VIEWSTATE" value="YTB6..." />
<script language="javascript"> <!--
  function __doPostBack(eventTarget, eventArgument) {
     var theform = document.ctrl2
     theform.__EVENTTARGET.value = eventTarget
     theform.__EVENTARGUMENT.value = eventArgument
     theform.submit()
  }
// --> </script>
    <h3>Enter name:
      <input name="txtName" id="txtName"
             type="text" value="Joe" />
    Personality: <select name="Personality"
                         id="Personality">
    <option value="extraverted">extraverted</option>
    <option value="introverted">introverted</option>
    <option selected value="in-between">in-between</option>
    </select>
    <input name="ctrl7" type="button" value="Submit"
           onclick="javascript:__doPostBack('ctrl7','')">
    <div id="Greeting">Hi Joe, you selctedin-between</div>
</form></body></html>
```

Figure 15.5: HTML generated using server-side events

Controls

ASP.NET provides two sets of server controls. HtmlControls use the same names and syntax as their HTML counterparts, and WebControls provide a more consistent programming model and a higher level of abstraction.

HtmlControls

HtmlControls are server-side representations of standard HTML elements

- Any HTML element in an ASPX page marked with the `runat=server` attribute will become an HTML control on the server
- All derive from `HtmlControl` class
- HTML elements with no distinguished server-side functionality (like div, span, etc.) are all represented as `HtmlGenericControl` instances

ASP.NET defines two parallel sets of server-side controls: the Html controls and the Web controls. Html controls are designed to have direct translations from standard HTML elements, which means that you can effectively take a page of standard HTML, add "runat=server" attributes to all of the controls, change the page extension to .aspx, and you now have an ASP.NET page with a complete set of server side controls. Figure 15.6 shows all of the HtmlControls available, and the corresponding HTML tags they map to. Many HTML elements that do not have distinct attributes that would be relevant to server-side programming (like span, div, body, font), so these elements are all grouped under the `HtmlGenericControl` that has a generic set of properties an methods that apply to all of these controls. Figure 15.7 shows a sample page written using server-side Html controls.

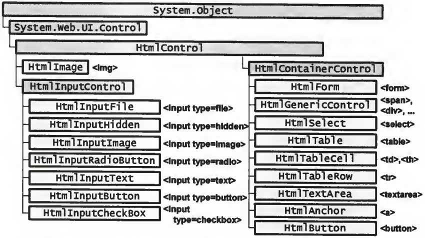


Figure 15.6: Hierarchy of HtmlControls and the tags they map to


```
<%@ Page Language="C#" %>
<html>

<body>
  <form runat=server>
  <input type=radio runat=server>click me</input><br/>
  <input type=checkbox runat=server>check me</input><br/>
  <input type=button value="Push me" runat=server /><br/>
  <input type=text value="type in me" runat=server /><br/>
  <textarea value="type more in me" runat=server /><br/>
  <table runat=server>
     <tr><td>cell00</td><td>cell01</td></tr>
     <tr><td>cell10</td><td>cell11</td></tr>
  </table>
  </form>
</body>
</html>
```

Figure 15.7: A sample ASP.NET page written with HtmlControls

WebControls

WebControls provide a more consistent object model and a higher level of abstraction than HtmlControls

- Most HTML elements can also be represented as `WebControls` on the server
- `WebControl` versions typically have a more consistent interface (background color is always `BackColor` property whereas in HTML it may be a style attribute (span) or a property (table))
- `WebControls` also provide higher-level controls with more functionality than primitive HTML elements (like the Calendar control)
- `WebControls` may render themselves differently based on client browser capabilities

The other set of control provided by ASP.NET are the `WebControls`. These controls duplicate much of the functionality of the `HtmlControls`, but provide a more consistent object model, as well as more complex controls. For example, to set the background color of a `table` element in HTML, you need to modify the `BGColor` property, but to change the background color in a `span` element, you must modify its style attribute. In contrast, all `WebControls` have the inherited property of `BackColor` that controls their background color. Many `WebControls` provide a richer, more complex interface than traditional HTML elements, like the `Calendar` control, which generates a complete calendar in the client browser. `WebControls` also may render themselves differently depending on the client's browser capabilities to accommodate as many clients as possible. Figure 15.8 shows the hierarchy of web controls available in ASP.NET, and figure 15.9 shows the graphic rendering of each control.

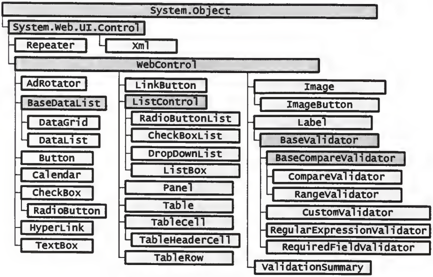


Figure 15.8: Hierarchy of WebControls

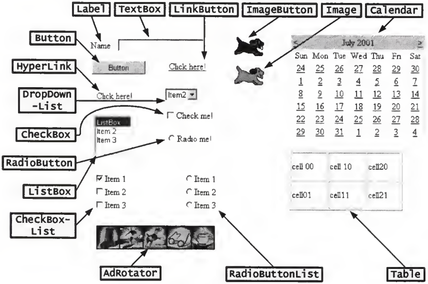


Figure 15.9: WebControl catalog

As an example of an ASP.NET page written using `WebControls` instead of `HtmlControls`, consider figure 15.10. Notice that all `WebControls` are wrapped in the `asp` namespace. It should also be noted that it only makes sense (and only works) if a `WebControl` is run with the `runat=server` attribute set.

```
<%@ Page Language="C#" %>
<html>

<body>
  <form runat=server>
  <asp:RadioButton Text="click me" runat=server/><br/>
  <asp:CheckBox Text="check me" runat=server/><br/>
  <asp:Button Text="Push me" runat=server /><br/>
  <asp:TextBox Text="type in me" runat=server /><br/>
  <asp:TextBox TextMode=MultiLine rows=3
             Text="type more in me" runat=server /><br/>
  <asp:Table runat=server>
    <asp:TableRow>
      <asp:TableCell>cell00</asp:TableCell>
      <asp:TableCell>cell01</asp:TableCell>
    </asp:TableRow>
    <asp:TableRow>
      <asp:TableCell>cell10</asp:TableCell>
      <asp:TableCell>cell11</asp:TableCell>
    </asp:TableRow>
  </asp:Table>

  </form>
</body>
</html>
```

Figure 15.10: A sample ASP.NET page written with WebControls

IE Web Controls

IE Web Controls Provide sophisticated DHTML rendering of advanced UI elements

- Provided as a separate add-on
- `TreeView`
- `Toolbar`
- `TabStrip`
- `MultiPage`

In addition to the Web controls that ship with the .NET runtime, there are a set of complex custom controls provided as a separate installation called the IE Web controls. There are a total of four controls, the `TreeView`, the `Toolbar`, the `TabStrip`, and the `MultiPage`. An image of each of these controls in action is shown in figure 15.11. The hierarchy for the classes defining these controls is shown in figure 15.12. Figure 15.13 shows an example of each of these controls being used in an ASP.NET page.

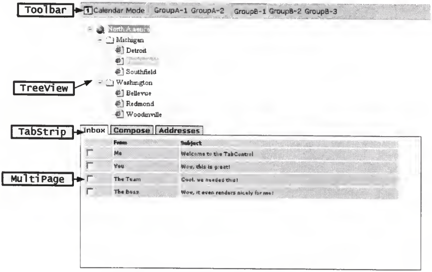


Figure 15.11: IE WebControl catalog

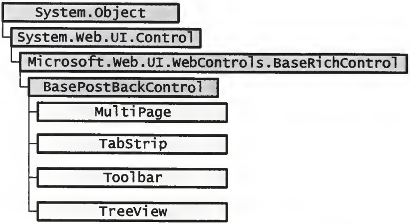


Figure 15.12: IE WebControls hierarchy

```
<%@ Register TagPrefix="iewc"
    Namespace="Microsoft.Web.UI.WebControls"
     Assembly="Microsoft.Web.UI.WebControls" %>
<html><body>
  <form runat=server>
<iewc:TabStrip id="TabStrip1" runat="server"
               TargetID="MultiPage1">
    <iewc:Tab Text="Shopping Cart" />
    <iewc:TabSeparator />
    <iewc:Tab Text="Receipt" />
    <iewc:TabSeparator DefaultStyle="width:100%" />
</iewc:TabStrip>

<iewc:MultiPage id=MultiPagel runat="server">
<iewc:PageView id="Cart"> <!-- other controls here-->
</iewc:PageView>
<iewc:PageView id="Receipt"><!-- other controls here-->
</iewc:PageView>
</iewc:MultiPage>

<iewc:Toolbar id=Toolbarl runat="server">
 <iewc:ToolbarButton Text="Open" ></iewc:ToolbarButton>
 <iewc:ToolbarButton Text="Save" ></iewc:ToolbarButton>
 <iewc:ToolbarButton Text="Close" ></iewc:ToolbarButton>
</iewc:Toolbar>
</form></body></html>
```

Figure 15.13: A sample ASP.NET page written with IE WebControls

Custom Controls

ASP.NET provides an extensible control framework for defining custom server controls. Custom controls are reusable, extensible, web components and are the core elements in an ASP.NET system.

Controls and Pages

Every ASP.NET Page is constituted by a collection of controls

- `System.Web.UI.Page` is itself the top-level control
- `System.Web.UI.HTMLControls` define server-side equivalents of HTML elements
- `System.Web.UI.WebControls` define server-side controls that generate HTML in an intuitive, standard way
- `System.Web.UI.LiteralControl` is the catch-all control used to represent all literal HTML text on a Page
- `System.Web.UI.Control` is the base class for all of these controls, and can be extended to build custom controls

Recall that an ASP.NET page consists of a collection of controls, including the page object itself, acting as the top-level control. A page is rendered to a client by recursively requesting that each control on the page render itself as HTML. An ASP.NET page may contain `HTMLControls`, which mirror their respective HTML elements, and render that element when requested. It also may contain `WebControls`, which are higher-level controls with a more uniform interface than native HTML elements, but which still render HTML when the page is rendered. Any literal HTML contained in an ASP.NET page is treated as a `LiteralControl`, which simply renders the literal HTML when requested. Figure 15.14 shows a simple .ASPX page as it sits on disk, and figure 15.15 shows the hierarchical control tree that will be generated by the compiled .ASPX page when a client requests it. Note that all elements of the page are turned into controls, even literal chunks of HTML are wrapped by a `LiteralControl`.

```
<%@ Page Language="C#" %>
<html><body>
<form runat=server>
  <h3>Enter name:
     <asp:textbox id="name" runat=server />
  </h3>
  <input type=submit value="Submit" />
  <p>
    <% if(IsPostBack) { %>
      Hi <%=name.Text%>! How are you?
     <%}%>
  </p>
  </form>
</body></html>
```

Figure 15.14: An .ASPX page prior to compilation

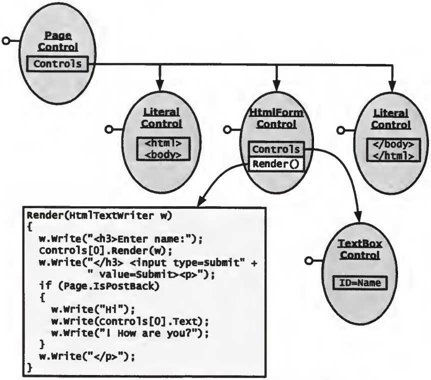


Figure 15.15: In-memory representation of an .ASPX page

All server controls in ASP.NET derive from the base class `System.Web.UI.Control`, which provides the necessary methods and events to define a control's behavior. Figure 15.16 shows a view of this class with the most commonly used methods and properties you will use as a control writer. The `Render()` method is perhaps the most important for server controls. When an ASP.NET page displays itself, it iterates across its controls, invoking the `Render()` method of each one. `Render()` takes a reference to an `HtmlTextWriter` class as a parameter, which should be used to write HTML. It has methods like `Write` and `WriteLine`, as well as helper functions like `WriteBeginTag` to aid in generating HTML.


```
public class Control : IComponent, IParserAccessor, ...
{
     // Called to render a control as HTML
  protected virtual void Render(HtmlTextWriter w);
     // Issued when a control is first created
  public event EventHandler Init;
     // Request to create child controls
  protected virtual void CreateChildControls();
     // Access to containing Page
  public virtual Page Page {get; set;}
     // Access to immediate parent control
  public virtual Control Parent {get;}
  //...
}
```

Figure 15.16: Important members of the System.Web.UI.Control class

Building Custom Controls

You build new custom server controls by creating a new class derived from Control

- Create a new class derived from `System.Web.UI.Control` (typically in a distinct namespace)
- Add any control-specific properties
- Add any control-specific events
- Override its `Render()` method to generate HTML using the `HtmlTextWriter` passed in
- Compile the code into an assembly and deploy

New server controls are created by creating new classes derived from the `System.Web.UI.Control` class. At the very least, you will typically override the `Render()` method to specify the HTML that should be generated when the control is requested to render itself. You may also decide to add control-specific properties and events. Figure 15.17 shows an example of a custom control that simply renders an `h1` tag displaying the value of its string property. Controls should not, in general, generate `html`, `body`, or `form` tags, as they are typically used within a client page already containing these elements.

```
namespace DM.AspDotNet
{
using System;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.ComponentModel;

  public class SimpleControl : Control
  {
     private string m_Prop = "";

     public string Prop
     {
       get { return m_Prop; }
       set { m_Prop = value; }
     }

     protected override void Render(HtmlTextWriter output)
     {
       output.Write("<h1>This is my control with property
value: ");
      output.Write(m_Prop + "</h1>");
     }
  }
}
```

Figure 15.17: A simple custom control

This code would be compiled into an assembly, and then deployed, either in the Global Assembly Cache, or in the /bin directory of the web application. Any assembly placed in the /bin directory of a web application will implicitly be referenced by all pages in that application. In order to successfully compile this control, you will have to reference System.dll and System.Web.dll, as shown in the following build command for a file named SimpleControl.cs:

```
csc /r:System.dll /r:System.Web.dll /t:library
/out:bin\SimpleControl.dll SimpleControl.cs
```

Using Custom Controls

Custom controls may be used from any ASP.NET page with the Register directive

- `<%@ Register TagPrefix="DM" Namespace="DM.AspDotNet" %>`
- `TagPrefix` is the shorthand name you would like to use to scope any controls in the `Namespace`
- `Namespace` specifies the namespace of the control you want to reference
- `Assembly` is the name of the assembly containing the control

To use a custom control in an ASP.NET page, you need to reference the control using the `@Register` directive. This directive takes the namespace in which the control lives, a `TagPrefix` that is a string alias to the namespace, and the name of the assembly containing the control definition. Figure 15.18 shows a complete .aspx page that uses the `SimpleControl` shown earlier.

```
<%@ Page Language="C#" %>
<%@ Register TagPrefix="DM.AspDotNet"
Namespace="DM.AspDotNet"
             Assembly="SimpleControl" %>

<html>
<body>

  <form runat=server>
    <DM.AspDotNet:SimpleControl id="MyCtrl" Prop="Hi!"
runat=server />
  </form>

</body>
</html>
```

Figure 15.18: Client .aspx page for the SimpleControl custom server control

An alternative to pre-compiling a custom control, is to include a reference to its source file as part of the `src` attribute of the `Page` directive. ASP.NET will take care of compiling the control code into the generated page assembly, and the `Register` directive remains unchanged.

Composite Controls

Custom controls can contain other embedded controls

- Involves adding a new control to the `Controls` collection
- Child controls should be created in the `CreateChildControls()` method
- Use the `LiteralControl` class to intermingle HTML tags with controls
- Composite controls should implement the `INamingContainer` interface to scope its child controls in a distinct namespace

A compelling reason to use custom controls in ASP.NET applications is the potential for reuse. By embedding other controls within them, custom controls can be used to define "chunks" of forms that can potentially be reused from many different pages, complete with their own properties, events, and methods. Composite controls are built by creating child controls and adding them to the `Controls` collection of the parent control. A special virtual function called `CreateChildControls` should be overridden. This function is from the `Control` base class, which will be called at the time when child controls should be created and added to the `Controls` collection.

Figure 15.19 shows a simple composite control that implements a calculator using three `TextBox` controls, a `Button` control, and some `LiteralControls` intermingled. A `LiteralControl` is a simple control that simply renders its `Text` property, and is especially useful when building composite controls for properly laying out the child controls. This control also shows an example of hooking up an event handler to a child control.


```
public class SimpleComposite : Control, INamingContainer {
  TextBox m_Operand1; TextBox m_Operand2; TextBox m_Result;

  private void OnCalculate(Object sender, EventArgs e) {
   int res = Convert.ToInt32(m_Operand1.Text) +
             Convert.ToInt32(m_Operand2.Text);
    m_Result.Text = res.ToString();
  }

  public int Result { get { return
Convert.ToInt32(m_Result.Text); }  }

  protected override void CreateChildControls() {
    m_Operand1 = new TextBox();
    m_Operand2 = new TextBox();
    m_Result   = new TextBox();

    Controls.Add(m_Operand1); Controls.Add(new
LiteralControl(" + "));
    Controls.Add(m_Operand2); Controls.Add(new
LiteralControl(" = "));
    Controls.Add(m_Result);   Controls.Add(new
LiteralControl("&lt;br/&gt;"));

    Button calculate = new Button();
    calculate.Text = "Calculate";
    calculate.Click += new EventHandler(this.OnCalculate);
    this.Controls.Add(calculate);
  }
}
```

Figure 15.19: Composite Control example

Notice also, that the control shown in figure 15.19 implements the `INamingContainer` interface. This is a marker interface (one with no methods) that indicates to the containing page that this control has child controls that may need to be in a separate namespace. If there is more than one instance of a composite control on a given page, it is important that the child controls of each composite control do not have ID clashes. In general, it is a good idea to implement the `INamingContainer` interface on any composite control.

User Controls

User controls provide a simpler way of defining composite controls

- Instead of writing a composite control entirely in code, you can take any .aspx page and turn it into a 'user control'
- User controls are defined in .ascx pages and use the `@Control` directive instead of `@Page`
- All controls on a user control page are compiled into a separate composite control

Like pages in ASP.NET, controls can also be defined mostly in code, mostly as tags on a page, or somewhere in-between. So far, we have looked at defining controls in code, but it is possible to define something called a "user control" using tags on a page. This is most convenient for composite controls, because it allows you to layout your control's HTML tags on a page, rather than programmatically. To define a user control, you create a page with a .ascx extension and use the `@Control` directive where you might normally use a `@Page` directive in a standard .aspx page. The `@Control` directive takes the same attributes as the `@Page` attributes. You then layout the controls you want to appear in your user control (be sure not to include html, body, or form tags, as these will be supplied by the client). You can also add properties and events in a server-side script block. Figure 15.20 shows our calculator control re-written as a user control. Notice that properties and methods are declared within the server-side script block as if we were inside of our control class definition. This .ascx page will be compiled into a distinct control class when referenced by a client .aspx page.

```
<%@ Control Description="This control provides a simple
calculator" %>

<asp:TextBox ID="Op1" runat=server/> +
<asp:TextBox ID="Op2" runat=server/> =
<asp:TextBox ID="Res" runat=server/>
<br/>
<asp:Button Text="Calculate" OnClick="OnCalculate"
runat=server/>

<script language="C#" runat=server>
  private void OnCalculate(Object sender, EventArgs e)
  {
    int res = Convert.ToInt32(Op1.Text) +
Convert.ToInt32(Op2.Text);
     Res.Text = res.ToString();
  }

  public int Result
  {
    get { return Convert.ToInt32(Res.Text); }
  }
</script>
```

Figure 15.20: A User Control

Using User Controls

Access to a user control is much like any other custom control

- Clients reference user controls using the `@Register` directive specifying the .ascx file in the optional `Src` attribute
- User controls can also be created programmatically by calling `LoadControl` with the .ascx filename
- If loaded programmatically, the type of the user control will be `filename_ascx`. For example, foo.ascx would generate a type of `foo_ascx`

User controls can be accessed from any .aspx page much like other custom controls. The one major difference, is that the .ascx file must be accessible by the client page, and it must be referenced using the `Src` attribute of the `@Register` directive. If you need to load a user control dynamically, you can use the `LoadControl` method of the `Page` class, which takes the file name of the user control. When a user control is loaded programmatically, the name of the new control class will be the name of the file containing the user control, replacing the '.' with an underscore (_). For example, a user control written in file Foo.ascx would generate a new control class of type `Foo_ascx` when loaded with `LoadControl`. Figure 15.21 shows an example of an .aspx page, which is referencing two user controls. The first one, `user1`, is loaded using the `@Register` directive. The second, `user2` is loaded programmatically using the `LoadControl` method of the `Page` class.

```
<%@ Register TagPrefix="uc1" TagName="UserControl1"
    Src="UserControl1.ascx" %>

<html>
  <script language="C#" runat=server>
  void Page_Load(Object sender, EventArgs e)
  {
    UserControl uc2 = LoadControl("UserControl2.ascx");
    ((UserControl2_ascx)uc2).Prop = 100;
    Page.Controls.Add(uc2);
  }
  </script>
<body>

<form runat=server>
  <uc1:UserControl1 id="UC1" runat=server/>
</form>

</body>
</html>
```

Figure 15.21: A sample client to a user control

Summary

- All pages in ASP.NET are compiled assemblies
- A page is constituted by collection of controls
- Pages are rendered by iteratively rendering all controls within them
- All controls derive from Control and typically override the Render() method
- Custom controls are referenced using the Register directive from an .aspx page
- Composite controls are controls that contain other controls as children
- User controls are composite controls authored in a .ascx page

ASP.NET User Content

are imp
using
the user
that access
id.

CLR thread Principal Info

Physical Token Info - can be imperson.
by setting a flag in web-config

Module 16

ASP.NET Security

This module examines the ASP.NET security infrastructure with an eye toward design. With all the different security contexts that come into play (managed principals, CAS, unmanaged process and thread tokens) it's really important to try to develop an intuition for how a managed web application should be configured. We recommend following the principal of least privilege, and describe how to use the authentication and authorization services provided by the infrastructure. Finally, we take a look explicitly at the forms-based authentication mechanism provided by ASP.NET; a technique very commonly used on websites.

After completing this module, you should be able to:

- enumerate the security contexts that affect an ASP.NET web application
- apply the principal of least privilege
- use ASP.NET to authenticate clients using native Windows authentication and Forms authentication
- restrict access to various pages using URL-based authorization and file system ACLs
- implement basic forms authentication in a web application
- add custom roles to forms authentication security contexts
- secure the cookies used by forms authentication

Introduction to ASP.NET Security

Security in ASP.NET can be somewhat tricky, because there are several security contexts you must consider, some managed, some unmanaged. This module explores the technologies that ASP.NET provides for authentication, authorization, and security context configuration.

Goals

Some Security Goals

- Principal of least privilege
- Authentication
- Authorization
- Fear (respect) user input

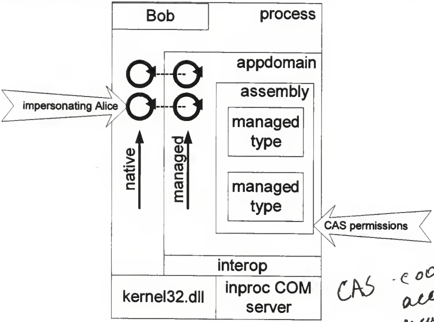
The principal of least privilege is a design trait that says code should run at the lowest possible level of privilege that allows it to complete its work. Authentication helps us figure out who is on the other end of the wire - clients often authenticate web servers, and web servers often authenticate clients. Authorization comes in after authentication - once you know who your client is, you can authorize them to do certain things, either via access control lists or via roles. Fearing user input is important, as this is generally the way your site will be attacked by a bad guy; bad guys often send finely crafted garbage as input to try to tickle bugs in your app, which may cause your app to do things it wasn't intended to do. Always assume input is evil until you've validated it.

Security Contexts

Code runs within several security contexts at once

- Unmanaged: process and thread tokens
- Managed: thread principal and CAS
- Unmanaged security context affects calls to unmanaged systems
- Managed security context affects your managed code
- Critical to understand and configure both correctly

Figure 16.1 shows the various security contexts that you should consider when designing managed types.



CAS - code access security

Figure 16.1: Security Contexts

The Worker Process

The security context of the worker process can be configured

- See <processModel> section of MACHINE.CONFIG
- User="Alice" Password="SomeCleartextPassword"
- User="SYSTEM" Password="Autogenerate"
- User="MACHINE" Password="Autogenerate"
- Latter is safest, uses built-in ASPNET account following principal of least privilege
- Can propagate thread token by setting <identity> in web.config

Figure 16.2 shows the worker process. The security context of the worker process can be configured via machine.config, and the security context of the thread can be configured via web.config (you either choose no thread token, or to propagate the token from the web server's thread).

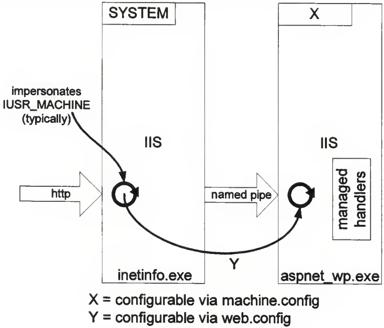


Figure 16.2: The Worker Process

Figure 16.3 shows the <processModel> attribute edited in machine.config such that the worker process runs under the ASPNET account. Figure 16.4 shows a web.config file that causes the security context of the thread in the web server to be propagated to the corresponding thread in the worker process.

```
<processModel
              ...
              User='MACHINE'
              Password='Autogenerate'
              ...
/>
```

Figure 16.3: Configuring the worker process security context


```
<configuration>
  <system.web>
    <identity impersonate='true'/>
  </system.web>
</configuration>
```

Figure 16.4: Propagating the thread security context

Use the principal of least privilege to choose these security contexts.

Client Authentication

ASP.NET provides several ways to authenticate clients

- Native Windows authentication – windows know the user
- Forms-based authentication – asp will auth.
- Passport authentication – ms.

Native Windows authentication passes the buck to IIS for client authentication, while Forms authentication allows you to build your own login. Passport authentication is designed to help you integrate with Microsoft's Passport single sign on.

Principals

IPrincipal and IIdentity

- Represents the result of authentication
- IIdentity indicates strength of authentication and a name
- IPrincipal binds an identity to a set of roles
- Context.User exposes the results
- Thread.CurrentPrincpal is also set after authentication

Figure 16.5 shows the IPrincipal and IIdenity interfaces.

```
interface IPrincipal {
  IIdentity Identity { get; }
  bool      IsInRole(string roleName);
}
interface IIdentity {
  bool   IsAuthenticated    { get; }
  string AuthenticationType { get; }
  string Name               { get; }
}
```

Figure 16.5: IPrincipal and IIdentity

Authorization

Authorization is all about controlling access to resources

- FileAuthorizationModule
- UrlAuthorizationModule
- Declarative Principal Checks
- Imperative Principal Checks

Authorization checks can be performed at many different granularities. File-based authorization is used automatically when you use native Windows authentication. Url-based authorization is configured via web.config, and an example is shown in figure 16.6. Note that all web.config files in the path from the target resource back up to the virtual root directory are considered, and the first match wins.

```
<configuration>
  <system.web>
    <authorization>
      <allow roles='Managers,Friends'/>
      <deny users='?'/>
    </authorization>
  </system.web>
</configuration>
```

Figure 16.6: Url-based authorization in web.config

Declarative principal checks are where you apply the PrincipalPermissionAttribute to classes or methods, allowing the CLR to check a role, user name, or authentication strength via Thread.CurrentPrincipal (see figure 16.7) Note that the wildcards '?' indicates unauthenticated users, and '*' indicates all users. Finally, imperative principal checks can be made anywhere in your code by instantiating a PrincipalPermission object and calling Demand(). Figure 16.8 illustrates this.

```
using System.Security.Permissions;

[PrincipalPermission(SecurityAction.Demand,
Authenticated=true)]
class PetStore {
  public void PetAnimals() { ... }
  public void BuyAnimals() { ... }

  [PrincipalPermission(SecurityAction.Demand,
Role="Staff")]
  public void FeedAnimals() { ... }

  [PrincipalPermission(SecurityAction.Demand,
Role="Admins")]
  public void GiveRaise() { ... }
}
```

Figure 16.7: Declarative principal demands


```
using System.Security.Permissions;

void VerifyIsManager() {
  IPermission p = new PrincipalPermission(null,
"Managers");
  p.Demand();
}
```

Figure 16.8: Imperative principal demands

Forms Authentication

The ASP.NET security framework includes support for forms-based authentication. This type of authentication has been used for years because of its portability and lowest common denominator approach.

Goals

Goals of Forms Authentication

- Use least-common-denominator technologies for portability
- Simple HTML forms for gathering authentication information
- Cookies for maintaining a session
- Flexible (e.g., password management)
- Extensible

Cookies

To understand Forms auth you must understand cookies

- Cookie mechanism documented in RFC 2965
- Web site sends state to user agent
- User agent echoes state back to server with each request
- User agent must not leak state across domains
- Cookies can be transient or persistent
- Nothing stops a client from modifying state in a cookie

Basics

Basic Forms Authentication

- Easy to get started with forms auth
- Turn it on in your root web.config file
- Design a login form that collects a user name and password/pin
- Upon submission, verify the password/pin and call FormsAuthentication.RedirectFromLoginUrl()

Figure 16.9 shows how to turn on forms authentication via web.config. Note that the entire <forms> element can be omitted if you are ok with the default values for the attributes (shown in the figure).

```
<configuration>
  <system.web>
    <authentication mode='Forms'>
      <forms name='ASPXAUTH'
             loginUrl='login.aspx'
             protection='All'
             timeout='30'
             path='/'
      />
    </authentication>
  </system.web>
</configuration>
```

Figure 16.9: Enabling Basic Forms Authentication

Figure 16.10 shows an example of a login page, and figure 16.11 shows the code that handles the submission.

```
<%@Page language='C#' %>
<%@ import namespace='System.Web.Security' %>

<form runat='server'>

<table><tr>
<td>Name:</td>
<td><asp:TextBox id='name' runat='server'/></td>
</tr><tr>
<td>Password:</td>
<td><asp:TextBox id='pwd' runat='server'/></td>
</tr></table>

<p><asp:CheckBox id='persist' runat='server'
  Text='Log me in automatically from this computer'/>

<p><asp:Label id='msg' runat='server'/>

<p><asp:Button Text='Login' runat='server'
  OnClick='OnLogin'/>

</form>
```

Figure 16.10: A Simple Login Form

```
<script runat='server'>
void OnLogin(Object sender, EventArgs eventArgs) {
  if (authenticateUserSomehow(name.Text, pwd.Text)) {
    // only redirect if password is valid
    FormsAuthentication.RedirectFromLoginPage(
         name.Text, persist.Checked);
  }
  else {
    // otherwise leave them on this page
    msg.Text = "Unknown user name or password";
  }
}
</script>
```

Figure 16.11: Handling the Login

Mechanics

Forms Authentication Mechanics

- FormsAuthenticationModule preprocesses all requests
- Cookie turned into IPrincipal and associated with context
- If no cookie and auth is required, redirects to login page
- After login, redirected back to original page, sending cookie
- Cookie contains version, name, timestamp, optional user data
- Cookie protected with encryption and MAC (by default)

A Message Authentication Code (MAC) is like an encrypted checksum. MAC protecting the cookie helps prevent the cookie from being changed by someone other than the web server, without being detected. This is important because the cookie is assumed to hold the authenticated user name, and clearly if this could be changed, the user could pretend to be anyone they wanted to if they could modify their cookie and get away with it.

Figure 16.12 shows how, during steady state operating, cookies are turned into security contexts (IPrincipal) by the FormsAuthenticationModule.

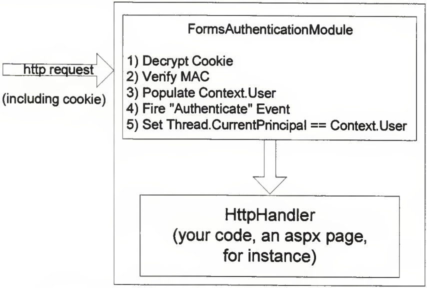


Figure 16.12: Transparent conversion of cookies to contexts

Figure 16.13 shows how the initial logon works. The request comes in without a cookie (or with a timed-out cookie, for instance), so the resulting security context is that of an anonymous user. The authorization module now takes over, and assuming that the requested resource requires authentication, generates a 401 Unauthorized response (note if the resource did not require authentication, the request would have been allowed through just fine). The FormsAuthenticationModule traps this 401 status code and converts it into a 302 redirect to the login page, which is what the user agent ultimately sees. Note that if this redirect occurs, the handler (your code) is NOT called for this request (the next activity you'll see in your code is on the login page).

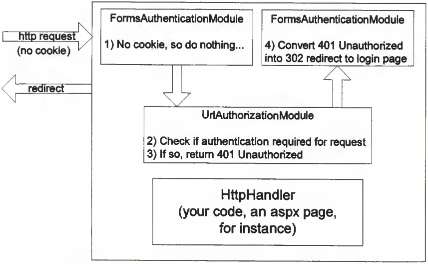


Figure 16.13: Transparent redirection to login page

Storing Passwords

Maintaining a password database

- Usually a good idea to maintain one way hashes of passwords instead of the cleartext passwords themselves
- Makes it more difficult to use a stolen password database
- You can maintain this database yourself
- You can use web.config as a simple password database
- Add a <credentials> child element to <forms>
- FormsAuthentication.CreatePasswordForStoringInConfigFile() calculates hashes for you
- FormsAuthentication.Authenticate() checks passwords

Figure 16.14 shows an example of a web.config file that hosts a password database, and figure 16.15 shows how to calculate the hash of a password (the value you store in the config file is a string of hexadecimal digits, by the way).

```
<configuration>
  <system.web>
    <authentication mode='Forms'>
      <forms loginUrl='MyLoginForm.aspx'>
        <credentials passwordFormat='SHA1'>
          <user name='Alice'
            password='9402F2262AB3B...'/>
          <user name='Mary'
            password='EA9003E959944...'/>
        </credentials>
      </forms>
    </authentication>
  </system.web>
</configuration>
```

Figure 16.14: Storing hashed passwords in web.config

```
using System;
using System.Web.Security;

class App {
  static void Main(string[] args) {
    Console.WriteLine(
    FormsAuthentication.HashPasswordForStoringInConfigFile(
      args[0], "sha1"));
  }
}
```

Figure 16.15: Creating hashed passwords for the config file

Figure 16.16 shows the code required to do a password lookup at runtime using FromsAuthentication.Authenticate(). This hashes the cleartext password you pass in, and compares the hash with what's in the web.config file for the specified user name. It's pretty straightforward.

```
using System.Web.Security;

bool authenticateUserSomehow(string name,
                             string cleartextPwd) {
  return FormsAuthentication.Authenticate(name,
                                          cleartextPwd);
}
```

Figure 16.16: Looking up Passwords

Postprocessing

Postprocessing after Authentication

- Often useful to do some additional work after FormsAuthenticationModule does its magic
- Add roles to principal...
- Convert user data in cookie to session state...
- To do any of these things, need to hook the AuthenticateRequest event
- Can do this via global.asax or by writing a module
- To add roles, just replace the principal!

Figure 16.17 shows an example global.asax, which postprocesses authentication in order to add roles. Another way to do this is to be lazy about role lookups - heck, the particular code path the request takes may never even cause a role check to occur, so why look it up if you don't have to? To do this, instead of looking up the roles immediately in AuthenticateRequest and using GenericPrincipal to carry those roles, implement IPrincipal yourself and only do the role lookup if IPrincipal.IsInRole() is actually called.

```
<%@application language='C#'%>
<%@import namespace='System.Security.Principal'%>

<script runat="server">
public override void Init() {
  // wire up our post-authentication handler
  AuthenticateRequest += new
EventHandler(onAuthenticateRequest);
}

void onAuthenticateRequest(object sender, EventArgs args) {

  // get the principal produced by forms authentication
  IPrincipal originalPrincipal = HttpContext.Current.User;

  if (null != originalPrincipal) {
    IIdentity id = originalPrincipal.Identity;

    // TODO: lookup real roles based on id.Name
    string[] roles = { "ClubMember", "Swimmer" };

    // replace the principal with a new one with roles
    HttpContext.Current.User = new GenericPrincipal(id,
roles);
  }
}
</script>
```

Figure 16.17: Adding Roles using global.asax

Protecting Cookies

The security of authentication cookies is weak unless they are protected

- Cookies can be undetectably altered by users unless protected by a MAC
- Cookies can be stolen by an eavesdropper if sent over a non-secure channel
- Persistent cookies can be stolen off a user's hard drive
- Use SSL to encrypt all traffic to your secure site
- Don't be tempted to use SSL only for part of your site
- Cookie paths don't work well with IIS due to case sensitivity

Summary

- Security is critical in web applications!
- Managed code cares about managed security contexts (principals and CAS)
- Unmanaged code cares about unmanaged security contexts (thread and process tokens)
- Develop an intuition
- Use the principal of least privilege
- Use authentication and authorization to control access to your site

Module 17

ASP.NET Web Services

HTTP, XML, and SOAP have emerged as the key standards for interoperation between components and services in internet-scale distributed systems. ASP.NET provides a reflection-driven infrastructure that integrates into IIS to provide highly automated support for building standards-based web services on the .NET platform.

After completing this module, you should be able to:

- understand where XML fits as a component technology
- understand how XML Schema affects XML
- understand how Web Services Description Language (WSDL) extends the XML Schema language to support operations and services
- understand how the .NET platform supports web service development

ASP.NET Web Services

HTTP, XML, and SOAP have emerged as the key standards for interoperation between components and services in internet-scale distributed systems. ASP.NET provides a reflection-driven infrastructure that integrates into IIS to provide highly automated support for building standards-based web services on the .NET platform.

What is a Web Service?

A web service is an endpoint that...

- Is accessible via a standard Internet communication protocol (HTTP, HTTPS, SMTP)
- Processes XML messages using SOAP
- Defines its message formats in terms of a portable type system (i.e., XML Schema)
- Provides metadata describing the messages it consumes and produces (i.e., WSDL)

17.1 17.2

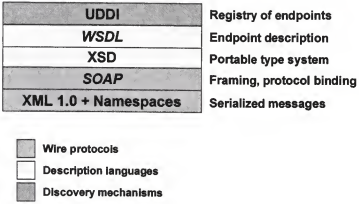


Figure 17.1: Web Service Protocols

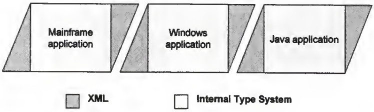


Figure 17.2: The Web Service Promise

What is SOAP?

SOAP is an XML protocol for message-based communication that defines...

- Format for message framing and extensibility
- Standard representation for errors
- Binding for sending messages over HTTP
- Binding for mapping messages to RPC invocations
- Rules for encoding common data types from programming languages (e.g.: C#, C++, Java, VB.NET, etc.)

17.3 17.4

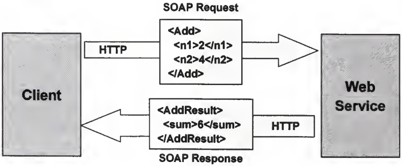


Figure 17.3: SOAP Request/Response

```
<soapv:Envelope
  xmlns:soapv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="http://www.develop.com/schemas/math"
>
  <soapv:Body>
    <ns1:Add>
      <n1>22</n1>
      <n2>33</n2>
    </ns1:Add>
  </soapv:Body>
</soapv:Envelope>
```

SOAP Frame

Payload

Figure 17.4: A SOAP Envelope

What is XML Schema?

XML Schemas are both an intrinsic type system for XML as well as a language for expressing user-defined types

- Schemas bring civilization to XML
- Brings civilization to XML
- Rich enough to handle most common type systems in use
- Schema processors provide validation
- Schema processors add reflection capabilities to the Infoset
- Schema compilers can simplify low-level XML coding through automatic code generation

17.5 17.6

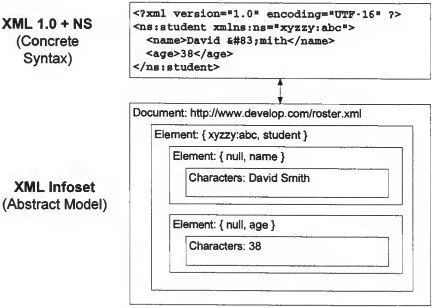


Figure 17.5: XML Without Schema

XML Schema

```
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="xyzzy:abc"
   targetNamespace="xyzzy:abc"
>
  <xsd:complexType name="person" >
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="age" type="xsd:double" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="student" type="tns:person" />
</xsd:schema>
```

Post-Schema XML Infoset
(Abstract Model)

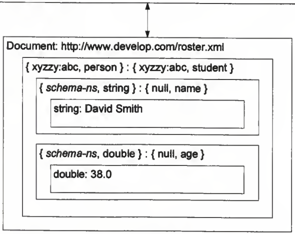


Figure 17.6: XML With Schema

The Web Service Description Language (WSDL)

In order to interact with a web service, clients need information not specified by SOAP

- Target URL
- Format for messages
- Whether literal instances of XSD types or SOAP encoding rules will be used
- WSDL provides a machine-consumable description

17.7

```
<wsdl:definitions
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:s0="http://www.develop.com/webservices/" >
  ...
  <wsdl:service name="Calc">
    <wsdl:port name="CalcSoap" binding="s0:CalcSoap">
      <soap:address
        location="http://acmemath.com/MathService.asmx"/>
    </wsdl:port>
  </wsdl:service>
  ...
  <wsdl:binding name="CalcSoap" type="s0:CalcSoap">
    <soap:binding
transport="http://schemas.xmlsoap.org/soap/http"
                  style="document" />
    <wsdl:operation name="Add">
      <soap:operation
        soapAction="http://www.develop.com/webservices/Add"
        style="document" />
      <wsdl:input><soap:body use="literal" /></wsdl:input>
      <wsdl:output><soap:body use="literal"
/></wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

Figure 17.7: WSDL

Building Web Services

The .NET platform provides several options for implementing web services

- Program directly against HTTP and XML APIs
- Leverage out-of-the-box ASP.NET web service infrastructure
- Hybrid approach

Web Services in the Raw

Framework classes support manual web service development

- `System.Xml` namespace for client/server XML processing
- `System.Net.WebRequest` for client request/response I/O
- `System.Net.Socket` and related classes for server-side connection management and I/O
- Threading, asynchronous I/O on server for concurrency
- Write your own WSDL files
- Expressive and powerful, but involves heavy lifting

System.Web.Services

ASP.NET and the `System.Web.Services` namespace automates SOAP handling

- Built on ASP.NET HTTP pipeline and XML stack
- SOAP request/response mapped onto method call/return
- Works with any class adorned with `[WebMethod]` attributes
- Target class "bound" to .ASMX HTTP endpoint
- HTTP request "context" available during method execution
- Automated, customizable WSDL generation
- Automated client-side proxy generation

17.8 17.9 17.10

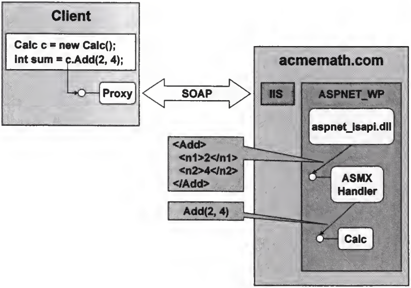


Figure 17.8: .NET Web Services Architecture

```
<%@ WebService language="C#" class="Calc" %>
using System.Web;
using System.Web.Services;

[WebService(Namespace="http://www.develop.com/webservices/"
)]
public class Calc : WebService
{
  [WebMethod]
  public int Add(int n1, int n2)
  {
    int sum = n1 + n2;
    HttpContext.Current.Application["last_sum"] = sum;
    return(sum);
  }

  [WebMethod]
  public int GetLastSum()
  {
    return
(int)HttpContext.Current.Application["last_sum"];
  }
}
```

Figure 17.9: Implementing a WebMethod in a .ASMX File


```
c:\> wsdl /language:cs /namespace:MathService /out:calcproxy.cs
     http://acmemath.com/MathService.asmx?wsdl
```

calcproxy.cs

```
namespace MathService {
  public class Calc : SoapHttpClientProtocol {
    public Calc() {
      this.Url = "http://acmemath.com/MathService.asmx";
    }
    public int Add( int n1, int n2 ) {
      object[] results = this.Invoke( "Add",
                                      new object[] {n1, n2} );
      return((int)results[0]);
    }
  }
}
```

client.cs

```
class Client {
  static void Main() {
    MathService.Calc calc = new MathService.Calc();
    Console.WriteLine("2 + 2 = " + calc.Add(2, 2));
  }
}
```

Figure 17.10: Using WSDL.EXE to Generate a Proxy

Processing Soap Headers

SOAP header processing is also highly automated

- Headers support protocol extensions over SOAP
- Headers may be optional or mandatory
- Service defines class derived from `SoapHeader` class
- Service defines public field using custom header class
- WebMethods use `SoapHeaderAttribute` to request header processing
- WSDL.EXE-generated proxy enables seemless client usage

17.11 17.12 17.13 17.14

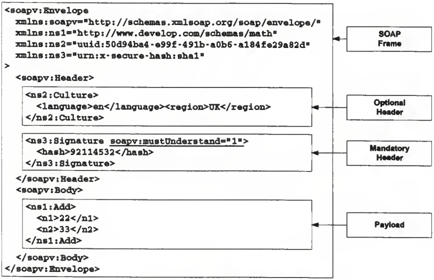


Figure 17.11: SOAP Envelope Headers

```
<%@ WebService language="C#" class="Calc" %>
using System.Web.Services;
using System.Web.Services.Protocols;

public class SignatureHeader : SoapHeader
{
    public int hash;
}

public class CultureHeader : SoapHeader
{
    public string language;
    public string region;
}

...
```

Figure 17.12: Defining SOAP Headers

```
<%@ WebService language="C#" class="Calc" %>

...

[WebService(Namespace="http://www.develop.com/webservices/"
)]
public class Calc : WebService
{
  public SignatureHeader signatureHeader;
  public CultureHeader   cultureHeader;

  [WebMethod]
  [SoapHeader("signatureHeader", Required=true)]
  [SoapHeader("cultureHeader", Required=false)]
  public int Add(int n1, int n2)
  {
    VerifyMessage(n1, n2, signatureHeader.hash);

    if( cultureHeader != null )
    {
      // Process optional culture header...
    }

    return(n1 + n2);
  }
}
```

Figure 17.13: Processing SOAP Headers

```
using MathService;

Calc calc = new Calc();
calc.signatureHeader = new SignatureHeader();
calc.signatureHeader.hash = ComputeHash(22, 33);
calc.cultureHeader = new CultureHeader();
calc.cultureHeader.language = "en";
calc.cultureHeader.region = "UK";
int sum = calc.Add(22, 33);
```

Figure 17.14: Setting SOAP Headers on Client

SOAP Extensions

SOAP extensions allow SOAP messages to be modified as they are transmitted between clients and services

- Interception-based model supports seemless feature integration
- Client- and service-side interception supported
- Extension class derives from `SoapExtension` class
- Extensions can be installed using configuration files
- Extensions can be installed using custom `SoapExtensionAttribute`
- Examples include compression, encryption, logging, etc.

17.15

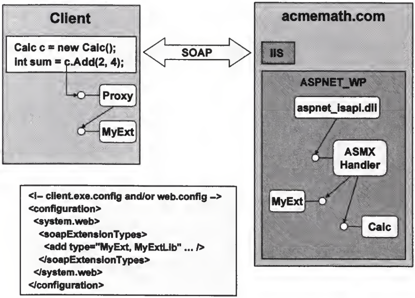


Figure 17.15: SOAP Extension Architecture

When WebMethods Fall Short

The WebMethod infrastructure does not meet every need of every web service

- Built-in SOAP/method-call mapping targets typical use-case
- Some services need alternative SOAP message processing
- Most services still want built-in concurrency, I/O management
- Custom `IHttpHandler` combined with XML APIs provides nice alternative
- Note: need to write your own WSDL files

17.16 17.17

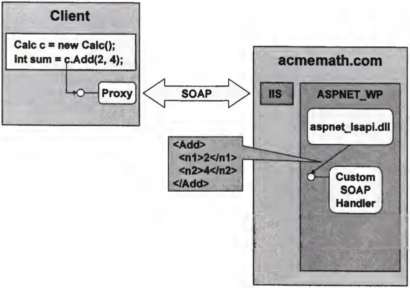


Figure 17.16: Custom SOAP Handlers

```
public class CustomSOAPHandler : IHttpHandler
{
  public void ProcessRequest(HttpContext ctx)
  {
    // Load input XML from HTTP request message body.
    XmlReader reader = new
XmlTextReader(ctx.Request.InputStream);
    XmlDocument xmlRequest = new XmlDocument();
    xmlRequest.Load(reader);

    // Process request to taste, writing response to
    // output stream to be transmitted back to client...
    //
    ctx.Response.ContentType = "text/xml";
    ProcessSoapRequest(xmlRequest, ctx.Response.Output);
  }

  public bool IsReusable { get { return(true); } }
}
```

Figure 17.17: A Custom SOAP Handler

Web Service Discovery

UDDI is a standard web service discovery protocol and repository

- Universal Description, Discovery, and Integration service
- Defined by Microsoft, IBM, and Ariba
- Industry consortium established (uddi.org)
- Protocol based on SOAP
- Supports attribute-based queries against desired characteristics
- Storage of metadata replicated to avoid centralization
- Supported by VS.NET

17.18

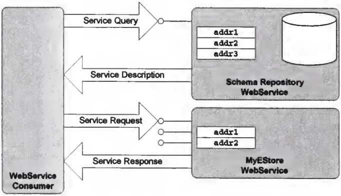


Figure 17.18: UDDI in Action

Summary

- XML is based on portable data representations
- XML Schema is based on portable type descriptions
- SOAP provides framing for XML Schema-based data
- WSDL maps representations to operations, endpoints, and services

Mano on linux

~~rotor~~? on free BSD

xmethods.net

http://staff.develop.com/woodring/gnet/may2002.zip

Module 18

XML Serialization

XML has its own type system that is based on XML Schemas (XSD). The System.Xml.Serialization library allows XSD types to be mapped to CLR types in a relatively customizable way. This module introduces the System.Xml.Serialization library and illustrates how it works.

After completing this module, you should be able to:

- describe the relationship between the XML and CLR type systems
- describe the role of XmlSerializer in System.Xml.Serialization
- use attributes to control how a CLR type is serialized to XML Schema
- customize the behavior of XmlSerializer

XML Serialization and Schemas

XML has its own type system that is based on XML Schemas (XSD). The System.Xml.Serialization library allows XSD types to be mapped to CLR types in a relatively customizable way.

XML vs. CLR type system

The XML type system is based on XML Schemas (XSD). The CLR type system is not.

- Ideally the two type systems are isomorphic
- In reality they are not
- SOAP section 5 attempted to identify 80/20 subset of both
- `System.Runtime.Serialization` dominated by programmatic types
- `System.Xml.Serialization` dominated by XML types

The XML Schema specification layers a representational type system over XML Infosets. XML Schema types have attribute inventory and content models, both of which augment the "value" of the type with named composite values. These named composite values are remarkably similar to the fields of a programmatic type, which causes programmers around the world to want to map back and forth between the two type systems, as shown in figure 18.1. As shown in this figure, if a mapping exists between a schema type and a programmatic type, then XML serialization can be derived automatically by reflecting against both type definitions.

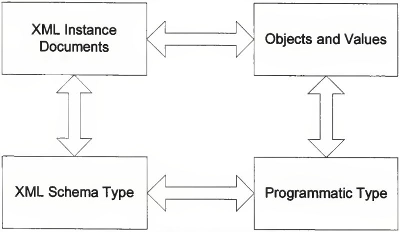


Figure 18.1: Bridging the XML type system

When mapping between the XML schema type system and a programmatic type system, issues invariably arise due to the impedance mismatch between in-memory vs. serialization concerns. Also, the XML schema type system needed to address the needs of SGML-style content management systems, which means that many XML schema constructs make little sense in a programmatic type system. These schisms lead to two fundamental approaches to bridging the two worlds: the XML Schema-centric approach and the programmatic type-centric approach. The former assumes that the XML Schema type system is the dominant type system, and that the concerns of the programmatic type system are subordinate. This approach is shown in figure 18.2. The latter assumes that the programmatic type system is the dominant type system, and that the concerns of the XML Schema type system are subordinate. This approach is shown in figure 18.3. The .NET framework supports both approaches by providing two XML-based serialization engines. The (`System.Runtime.Serialization`) serializer takes the programmatic type-centric view, and can take literally any type in the CLR and map it to the XML Schema type system with full fidelity. The (`System.Xml.Serialization`)

serializer takes the XML Schema-centric view, and can take a broader range of XML Schema types and map them to the CLR type system with reasonable fidelity. At the time of this writing, the `System.Runtime.Serialization` serializer was considerably more robust and closer to production quality than the `System.Xml.Serialization` serializer.

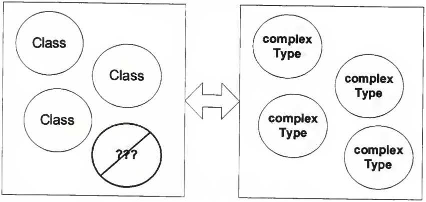


Figure 18.2: Schema-centric type mapping

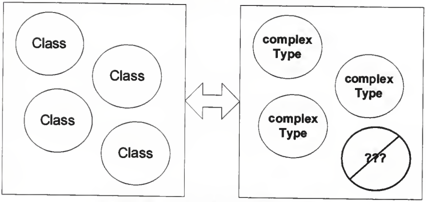


Figure 18.3: Class-centric type mapping

Neither the XML Schema-centric or type-centric approach is ideal, as compromises are required in both directions to ensure smooth interoperability. Issues such as typed references and arrays are not handled all that well in XML Schema. Issues such as mixed content, deriving from `int`, and optional values are not handled all that well in most programming systems. Until these issues are fully worked out, programmers will have a busy future mapping these mismatches by hand.

xsd.exe can take
metadata and convert it into schema.

XmlSerializer architecture

`XmlSerializer` is the focal point of the `System.Xml.Serialization` library

- `XmlSerializer` objects are bound to specific CLR types
- `Serialize` method writes an object (graph) through an `XmlWriter`
- `Deserialize` method reads an object (graph) from an `XmlReader`
- Both methods use underlying `XmlSerializationReader/Writer` types
- Reader/Writer code is generated into a dynamic assembly for speed

`XmlSerializer` is most developer's first exposure to the `System.Xml.Serialization` library. The `XmlSerializer` type provides a simple and convenient interface to the library's underlying serialization engine. `XmlSerializer` manages the generation and execution of per-type reader/writer pairs. These reader/writer pairs are actually dynamically generated types that extend `XmlSerializationReader` and `XmlSerializationWriter`, respectively. When instantiating an `XmlSerializer`, you must provide a `System.Type` object that represents the type the new serializer object will be used with. To avoid generating redundant assemblies, the `XmlSerializer` constructor looks in an AppDomain-wide cache of generated reader/writer assemblies and reuses the cached assembly if one is found. If a cached assembly is not found, the `XmlSerializer` constructor generates a new dynamic assembly by reflecting against the presented `System.Type` object. This new assembly is then added to the cache so that subsequent `XmlSerializer` objects can reuse it, reducing overall code size and codegen overhead.

As shown in figure 18.4, every `XmlSerializer` object holds a reference to a dynamically generated assembly. In most cases, this assembly is simply fetched from a cache and may in fact be shared by other `XmlSerializer` objects that are supporting the same CLR type. Figure 18.5 shows the signature of the `XmlSerializer` type. As this figure shows, the `XmlSerializer` type has exactly two methods: `Serialize` and `Deserialize`. The `Serialize` method simply invokes a well-known method on the generated `XmlSerializationWriter`-derived type. The `Deserialize` method invokes a different well-known method on the generated `XmlSerializationReader`-derived type. Figures 18.6 and 18.7 show the canonical usage of these two methods. Note that in these examples, the preferred overload that accepts an `XmlReader` or `XmlWriter` is not used, as these overloads do not work properly under Beta 1 of the .NET Framework.

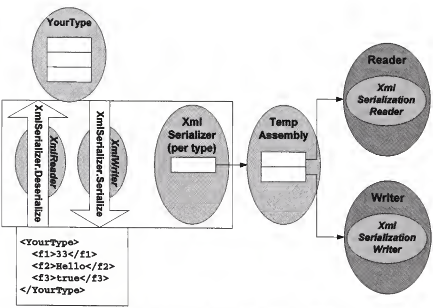


Figure 18.4: XmlSerializer object model

```
namespace System.Xml.Serialization {
  public class XmlSerializer {
    public XmlSerializer(Type clrType);
    public void Serialize(XmlWriter writer, object obj);
    public void Serialize(TextWriter writer, object obj);
    public void Serialize(Stream writer, object obj);
    public object Deserialize(XmlReader reader);
    public object Deserialize(TextReader reader);
    public object Deserialize(Stream reader);
  }
}
```

Figure 18.5: XmlSerializer (excerpt)


```
using System.IO;
using System.Xml;
using System.Xml.Serialization

static void WriteObjectToFile(Object obj, string filename)
{
  Stream writer = new FileStream(filename,
FileMode.Create);
  try {
    XmlSerializer ser = new XmlSerializer(obj.GetType());
    ser.Serialize(writer, obj);
  }
  finally {
    writer.Close();
  }
}
```

Figure 18.6: XmlSerializer.Serialize example

```
using System.Stream;
using System.Xml;
using System.Xml.Serialization

static object ReadObjectFromFile(Type type, string
filename) {
  Object result = null;
  Stream reader = new FileStream(filename, FileMode.Open);
  try {
    XmlSerializer ser = new XmlSerializer(type);
    result = ser.Deserialize(reader);
  }
  finally {
    reader.Close();
  }
  return result;
}
```

Figure 18.7: XmlSerializer.Deserialize example

Type mapping

Each CLR type may be affiliated with an XML Schema type using custom attributes

- **Default behavior assumes same-name `<xsd:complexType/>` per class/struct**
- `[XmlType]` **controls `<xsd:complexType/>` declaration**
- `[XmlRoot]` **controls global `<xsd:element/>` declaration**
- **Can suppress XML support altogether using** `[XmlType(IncludeInSchema=false)]`
- **The `CodeExporter` and `SchemaExporter` classes translate between schemas and CLR types**
- `CodeExporter/SchemaExporter` **wrapped by `XSD.EXE` tool**

The `XmlSerializer` architecture assumes that there is an XSD complex type for each CLR struct and class. Assuming no special attributes are used, this complex type is assumed to have one element child per public property or field. The name of the child element matches the field/property name. The type of the child element matches the field's/property's corresponding Schema type. All of the element children are assumed to be in an `<xsd:all/>` compositor.

The `[XmlType]` attribute allows you to control the name and namespace URI of the schema type that is associated with a CLR type. As shown in 18.8, the `[XmlType]` accepts a `Namespace` parameter that corresponds to the XML Schema's `targetNamespace` attribute. If this parameter is not present, then the corresponding schema type is not affiliated with any XML namespace. The `TypeName` parameter allows you to control the local name of the corresponding XML schema type, and if not provided, the name of the CLR type is used. Finally, to prevent any XML schema mapping from taking place, one can specify the `IncludeInSchema` parameter, which suppresses any XML schema conversion of the type.

has to be public

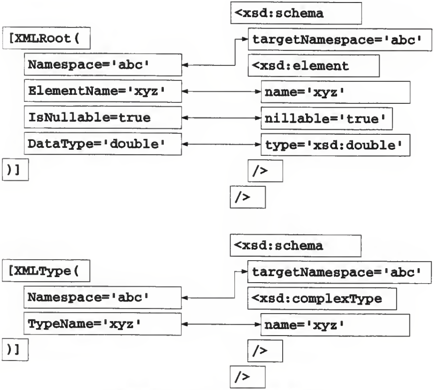


Figure 18.8: XmlRootAttribute mapping

If an XML Schema complex type is to be used as a "top-level" element in an XML instance document, then a global element declaration is also needed. The `[XmlRoot]` attribute causes such an element declaration to be created. As shown in figure 18.8, the `[XmlRoot]` attribute allows you to control the element name and XML namespace URI. It also allows you to control the `nillable` attribute for supporting `xsi:nil` in instance documents. Finally, if the element declaration use a built-in simple data type rather than a complex type, the `DataType` attribute can be used. Figure 18.9 shows each of the parameters to the `[XmlType]` and `[XmlRoot]` attributes.

Use	Parameter	Type	Description
R	ElementName	String	xsd:element/@name
T	TypeName	String	xsd:complexType/@name
R/T	Namespace	String	../xsd:schema/@targetNamespace
R	IsNullable	Boolean	xsd:element/@nillable
R	DataType	String	XSD Built-in Type Name
T	IncludeInSchema	Boolean	No corresponding XML type

Figure 18.9: [XmlRoot]/[XmlType] parameters

Using the `[XmlType]` and `[XmlRoot]` attributes is relatively simple. Consider the C# code shown in figure 18.10. The XML schema document that this code corresponds to is shown in figure 18.11. Note that because the CLR type `Don` does not have an `[XmlRoot]` attribute, there is no corresponding global element declaration in the XML schema document. Also note that because the CLR type `Steve` does not have an `[XmlType]` attribute, the corresponding complex type definition simply uses the CLR type name verbatim.

```
using System.Xml.Serialization

[ XmlRoot(ElementName="Bobby", Namespace="abcdef") ]
[ XmlType(TypeName="Robert", Namespace="abcdef") ]
public class Bob {}

[ XmlRoot(ElementName="Stevie", Namespace="abcdef") ]
public class Steve {}

[ XmlType(TypeName="Donald", Namespace="abcdef") ]
public class Don {}
```

Figure 18.10: [XmlRoot]/[XmlType] example


```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:target="abcdef"
  targetNamespace="abcdef"
>
  <xsd:complexType name="Robert" />
  <xsd:element name="Bobby" type="target:Robert" />

  <xsd:complexType name="Steve" />
  <xsd:element name="Stevie" type="target:Steve" />

  <xsd:complexType name="Donald" />
</xsd:schema>
```

Figure 18.11: [XmlRoot]/[XmlType] example output

The `System.Xml.Serialization` library can automatically generate XML schemas from a .NET assembly. It can also generate annotated source code from an XML schema. This capability is typically accessed using the `XSD.EXE` tool. As shown in figure 18.12, the `XSD.EXE` tool accepts either an assembly or an XML schema and uses the rules just described to generate the type definitions in the desired type system.

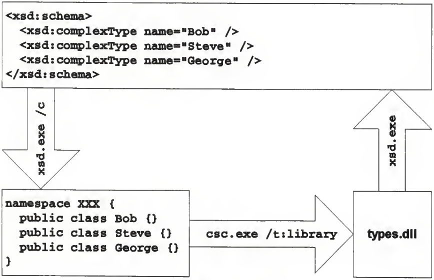


Figure 18.12: XSD.EXE

Schema customization

The underlying XML Schema format is controlled using custom attributes on the type and its fields/properties

- Defaults to same-named local `<xsd:element/>` for each public field/property
- `[XmlElement]` maps field/property to local `<xsd:element/>` declaration
- `[XmlAttribute]` maps field/property to local `<xsd:attribute/>` declaration
- `[XmlText]` maps field/property to characters in mixed content
- `[XmlEnum]` maps individual `enum` members to XML names
- `[XmlElement]` supports XSD `nillable` and `form`
- Multiple `[XmlElement]`s are used to specify legal substitutions (bad)

When mapping a CLR type to an XML schema type, each CLR field/property maps to some construct in the corresponding XML schema type. By default, all public fields and properties are assumed to map to child elements inside an `<xsd:all/>` compositor. This default mapping can be overridden using the `[XmlElement]` and `[XmlAttribute]` custom attributes. These attributes cause the attributed field/property to map to either a local element declaration or a local attribute declaration, respectively. Both of these attributes accept parameters that customize the generated XML schema types. These parameters are shown in figure 18.13.

Use	Parameter	Type	Description
E	ElementName	String	xsd:element/@name
A	AttributeName	String	xsd:attribute/@name
E/A	Form	XmlSchemaForm	@form
E/A	Namespace	String	@ref (combined with XXXName)
E/A	DataType	String	XSD Built-in Type Name
E/A	Encoding	"base64"|"hex"	Deprecated by XSD base64Binary/hexBinary
E	IsNullable	Boolean	xsd:element/@nillable
E	Type	Type	Explicitly allowed derived types

Figure 18.13: XmlAttributeAttribute/XmlElementAttribute parameters

The `[XmlElement]` attribute maps to a local element declaration in the XML schema language. As shown in figure 18.14, there are two usage models. The first model assumes a "pure" local element declaration, in which the namespace affiliation is controlled by the parent complex type. The second model assumes that a reference to a global element declaration will be used, and is typically only used to support element substitution groups in the XML schema language. As shown in figure 18.15, the `[XmlAttribute]` attribute works much the same way, except for XML attributes.

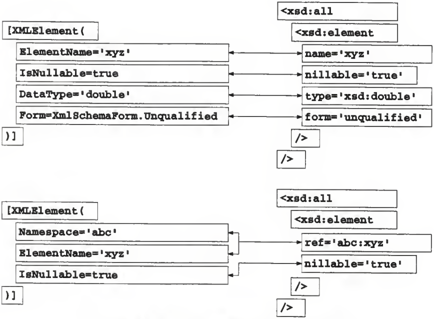


Figure 18.14: XmlElementAttribute mapping

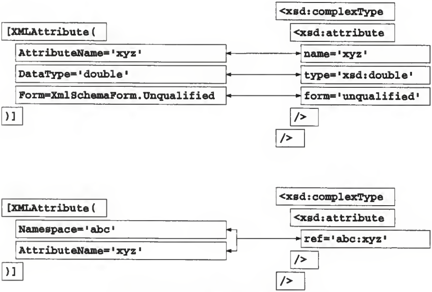


Figure 18.15: XmlAttribute mapping

Figure 18.16 shows an annotated CLR type that uses both the `[XmlElement]` and `[XmlAttribute]` attributes. The corresponding XML schema for this type is shown in figure 18.17. Note that in this example, the `rate` element is marked `form="qualified"` but the `notbreathing` element is not. This is due to the use of the `Form` parameter in the `[XmlElement]` attribute used in the C# code.

```
using System.Xml.Serialization
[ XmlType(TypeName="writer", Namespace="abcdef") ]
public class Author {
  [ XmlAttributeAttribute("name") ]
  public string name;

  [ XmlElementAttribute("rate", Form=XmlForm.Qualified) ]
  public double royaltyrate;

  [
    XmlElementAttribute("notbreathing",
                        Form=XmlForm.Unqualified,
              IsNullable=true)
  ]
  public bool dead;
}
```

Figure 18.16: [XmlAttribute]/[XmlElement] example

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:target="abcdef"
  targetNamespace="abcdef"
>
  <xsd:complexType name="writer" >
    <xsd:all>
      <xsd:element name="rate"
                type="xsd:double"
                form="qualified" />
      <xsd:element name="notbreathing"
                type="xsd:boolean"
                nillable="true"
      />
    </xsd:all>
    <xsd:attribute name="name" type="xsd:string" />
  </xsd:complexType>
</xsd:schema>
```

Figure 18.17: [XmlAttribute]/[XmlElement] example output

CLR enums are treated specially when mapping to XML schemas. A CLR enum maps to an XML schema simple type that restricts the built-in type `string` to a set of enumerated values, one per CLR enum member. The values of these restrictions correspond to the CLR enum member name unless the `[XmlEnum]` attribute is used. Figure 18.18 shows an annotated CLR enum and figure 18.19 shows the corresponding XML schema type.

```
using System.Xml.Serialization

[ XmlType(TypeName="hair", Namespace="abcdef") ]
public enum AuthorHair  {
  [ XmlEnum("bald") ] Harrisonish,
  [ XmlEnum("short")] Gudginy,
  [ XmlEnum("long")] Boxed,
  [ XmlEnum("longer") ] Rectoresque
}
```

Figure 18.18: [XmlEnum] example

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:target="abcdef"
  targetNamespace="abcdef"
>
  <xsd:simpleType name="hair" >
    <xsd:restriction base="xsd:string" >
      <xsd:enumeration value="bald" />
      <xsd:enumeration value="short" />
      <xsd:enumeration value="long" />
      <xsd:enumeration value="longer" />
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Figure 18.19: [XmlEnum] example output

At the time of this writing, one of the weaker aspects of the mapping between the CLR and XML type systems is related to substitution and inheritance. Both type systems have their own model for derivation and substitution, however, the current implementation of the serialization engine cannot always deal with these models in a reasonable way. The XML schema type system supports two forms of substitution. One form uses the `xsi:type` attribute to indicate that a derived type is in use. This form is supported by the serialization engine provided no `<xsd:sequence/>` compositors are in use. The other form, based on element substitution groups, is only partially supported, as the corresponding CLR type's field must be annotated with an list of all allowable substitutions.

Arrays

Fields/properties that are arrays need special treatment for array elements

- Arrays always serialized as element/child elements combination
- `[XmlElement]` functionality spread over `[XmlArray]` / `[XmlArrayItem]`
- `[XmlArray]` controls "container" element
- `[XmlArrayItem]` controls "item" elements
- Both support `Form`, `IsNullable`, `ElementName`, `Namespace`
- Multiple `[XmlArrayItem]`s are used to specify legal substitutions (bad)

When mapping fields/properties of array type, the `System.Xml.Serialization` library always maps the field/property to a local element declaration. The name of the element is controlled using the `[XmlArray]` attribute. The field/property's local element declaration contains an anonymous complex type definition which contains a compositor marked `minOccurs="0"` and `maxOccurs="unbounded"`. Inside that compositor resides another local element declaration that corresponds to the elements of the array. By default, the name and type of this inner local element declaration matches the CLR type of the array elements. One can control this mapping using the `[XmlArrayItem]` attribute.

For arrays whose element type is polymorphic, one can typically rely on the `xsi:type` functionality of XML schemas to be used. For more exotic uses, one can specify an a poor-man's element substitution group using the `[XmlArrayItem]` attribute as shown in figure 18.20. Note that in this example, the generated schema allows different element name/types in the inner-most compositor, as shown in figure 18.21.

```
using System.Xml.Serialization

[ XmlType("company", Namespace="abcdef") ]
public class Company  {
  [ XmlArray("staff",
            Form=XmlForm.Unqualified) ]
  [ XmlArrayItem("manager",
                typeof(Manager),
          Form=XmlForm.Unqualified) ]
  [ XmlArrayItem("clerk",
                typeof(Clerk),
          Form=XmlForm.Unqualified) ]
  public IEmployee[] Employees;
}
```

Figure 18.20: [XmlArrayItem] example

```
<xsd:complexType name="company" >
 <xsd:all>
  <xsd:element name="staff">
   <xsd:complexType>
    <xsd:all>
     <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="clerk" type="target:Clerk" />
      <xsd:element name="manager" type="target:Manager" />
     </xsd:choice>
    </xsd:all>
   </xsd:complexType>
  </xsd:element>
 </xsd:all>
</xsd:complexType>
```

Figure 18.21: [XmlArrayItem] example output

Customizing XmlSerializer

The `XmlSerializer` class supports several options to tailor its behavior at runtime

- `XmlSerializer` constructor accepts optional `XmlAttributeOverrides` to inject/override serialization attributes for pre-existing classes
- `XmlSerializer` fires `UnknownAttribute` events for unrecognized attributes
- `XmlSerializer` fires `UnknownNode` events for unrecognized element content (text, PIs, elements)
- Can pre-declare namespaces using extra parameter to `Serialize` to beautify/compact output

The XmlSerializer type supports a variety of optimizations/customizations. In particular, one can provide a set of custom attributes (e.g., [XmlType], [XmlElement]) when initializing an XmlSerializer. These attributes override any attributes that may have already been applied to the CLR type. The primary utility of this feature is that it allows you to layer the XML serialization attributes onto pre-existing classes that do not know about the XML serialization engine.

When deserializing, the XmlSerializer ignores unrecognized attributes, elements, character children, and processing instructions. For all but the last of these, that means that the deserializer will accept input documents that are not schema-valid. To allow developers to customize this behavior, the XmlSerializer type supports two events: UnknownAttribute and UnknownNode. The former is fired whenever a attribute is encountered in the input document that is not mapped to a corresponding CLR field or property. The latter is fired whenever a child is encountered in the input document that is not mapped to a corresponding CLR field or property. This includes processing instructions and ignorable whitespace, both of which do not affect schema validity. Figure 18.22 shows the usage of the UnknownAttribute event.

```
using System.Xml.Serialization

public static void OnAttribute(Object sender,
                               XmlAttributeEvent args) {
  string localName = args.Attr.ID.Name;
  string nsuri = args.Attr.ID.Namespace;
  string msg = String.Format(
                  "Unrecognized attribute {1}#{0}",
          localName, nsuri);
  throw new Exception(msg);
}

public static object ReadObject(XmlSerializer ser,
                               XmlReader reader) {
  ser.UnknownAttribute +=
                new XmlAttributeEventHandler(OnAttribute);
  return ser.Deserialize(reader);
}
```

Figure 18.22: UnknownAttribute handler

Finally, during serialization, namespace declarations are emitted as they are needed. This means that if a particular namespace URI is used several times in the output document, there will be several redundant namespace declarations. This makes the output document correct albeit quite verbose. When serializing, you can pass a collection of namespace declarations to the Serialize

method. This collection will be used to pre-declare namespaces at the root of the output document, suppressing the need for extra declarations in child elements.

System.Xml.Schemas

The "schema for schemas" is pregenerated into a CLR object model

- Provides a type-safe DOM-like model for schema traversal/generation
- Supports schema compilation to speed up validation and schema processing
- The `XmlSchema` class is focal point
- Watch for churn while .NET catches up to final XSD spec

The `System.Xml.Serialization` library provides a type-safe in-memory DOM for XML schema documents. Unfortunately, at the time of this writing, the implementation is far behind the XML schema specification, which means that most/all of the library will be impacted by at least minor modifications over the next few months. Anticipated features of the library include schema compilation and general-purpose validation, neither of which are working in Beta 1.

Summary

- The XML type system is based on XML Schema while the CLR type system is not
- XmlSerializer is the focal point of the System.Xml.Serialization library
- Each CLR type may be affiliated with an XML Schema type using custom attributes
- The XmlSerializer class supports several options to tailor its behavior at runtime
- The "schema for schemas" is pregenerated into a CLR object model

Reflection tools .

reflector

Lutz Roeder's .Net Reflector (reflector.exe)

Anakrino.exe / open assembly, look at il

and decompile the assembly back into c#

www gotdotnet

fxcop: w... /team/libraries

Anakrino: ... /team/csharp/ThirdParty

Can use:
using DWORD = System.UInt32;

Module 19

Win32/COM Interoperation

Componant.msc

Code written for .NET's Common Language Runtime rarely can stay in the runtime forever. Fortunately, the CLR provides an interoperation layer that allows managed code to call out to native DLLs and COM components as well as allowing managed objects to be exported as COM objects.

After completing this module, you should be able to:

- call into classic Win32 DLLs
- call into classic COM components
- export CLR types to classic COM and script

Interop Basics

Code written for .NET's Common Language Runtime rarely can stay in the runtime forever. Fortunately, the CLR provides an interoperation layer that allows managed code to call out to native DLLs and COM components as well as allowing managed objects to be exported as COM objects.

Motivation

Not all code can execute in the runtime

- "Heritage" source code can be translated (port)
- "Heritage" DLLs can be integrated (punt)
- CLR types can plug into COM-based frameworks
- The runtime can be hosted in your container

Despite the hopes of management, shareholders or overly-ambitious developers, it is simply not possible to get every piece of software ever written to run as managed code in .NET's Common Language Runtime. Given the shortage of programming talent, it is unlikely that many organizations will rewrite 100% of their existing Win32/C/C++/VB6/COM codebase no matter how compelling the new runtime may be. That means that the world of native code written before the .NET era needs to peacefully coexist with new managed components written in .NET-friendly languages. Thankfully, the CLR supports bridging the two worlds without undue pain or suffering.

To allow native and managed code to coexist, some mechanism is needed to allow managed code executing inside of the CLR (MSCOREE.DLL) to integrate with code running outside of MSCOREE.DLL. This means that managed programs need to run in "unmanaged mode" long enough to execute classic Win32 or COM code. Also, unmanaged threads may need to visit MSCOREE.DLL long enough to invoke a method on a managed type. Fortunately, the CLR supports both types of transitions, allowing both managed and native DLLs to coexist in a single process and call from one DLL to another independent of whether or not they are hosted inside the CLR. An example of a typical circa-2001 process is shown in figure 19.1. Note that some code runs inside of MSCOREE.DLL and some does not. Unless the entire operating system is rewritten in managed code, this is likely to be the state of affairs for some time, as today, the underlying Windows operating system is exposed via native DLLs, not managed DLLs.

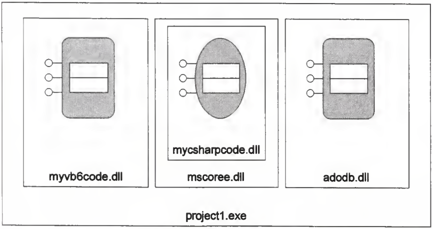


Figure 19.1: Integrating managed and unmanaged code

P/Invoke

P/Invoke is the technology that allows managed code to call out to unmanaged code

- LoadLibrary/GetProcAddress automated using `static` methods marked `extern`
- Can control all aspects using `DllImport` attribute
- No call to `LoadLibrary` until first invocation (a la delay load)
- Unicode/ANSI issues handled explicitly or automatically based on platform

P/Invoke is used for static methods

Calling a native Win32 DLL from the CLR is fairly straight-forward using P/Invoke (the "P" stands for "platform"). P/Invoke is a technology that allows you to map a static method declaration to a PE/COFF entry point that is resolvable via `LoadLibrary`/`GetProcAddress`. Like Java Native Interface (JNI) and J/Direct before it, P/Invoke uses a managed method declaration to describe the stack frame, but assumes the method body will be provided by an external, native DLL. Unlike JNI, however, P/Invoke is useful for importing "heritage" DLLs that were not written with the CLR in mind.

To indicate that a method is defined in an external, native DLL, you simply mark the static method as `extern` and use the `System.Runtime.InteropServices.DllImport` method attribute. The `DllImport` attribute informs the CLR which arguments to pass to `LoadLibrary` and `GetProcAddress` when it is time to call the method. The built-in C# `dllimport` attribute is simply an alias for `System.Runtime.InteropServices.DllImport`.

The `DllImport` attribute takes a variety of parameters. As shown in figure 19.2, the `DllImport` attribute requires at least a file name to be provided. This file name is used by the runtime to call `LoadLibrary` prior to dispatching the method call. The string to use for `GetProcAddress` will be the symbolic name of the method unless the `EntryPoint` parameter is passed to `DllImport`. Figure 19.3 shows two ways to call the `Sleep` method in `kernel32.dll`. The first example relies on the name of the C# function matching the name of the symbol in the DLL. The second example relies on the `EntryPoint` parameter instead.

Parameter Name	Type	Description	Default
Value	System.String	Path for LoadLibrary	<mandatory>
EntryPoint	Syetem.String	Symbol for GetProcAddress	<methodname>
CallingConvention	CallingConvention	Stack cleanup/order	Winapi
CharSet	CharSet	WCHAR/CHAR/TCHAR	Ansi
ExactSpelling	System.Boolean	Don't look for name with A/W/@	false
PreserveSig	Syetem.Boolean	Don't treat as [out,retval]	true
SetLastError	System.Boolean	GetLastError valid for call	false

Figure 19.2: DllImport attribute parameters

```
using System.Runtime.InteropServices;

public class K32Wrapper {
  [ DllImport("kernel32.dll") ]
  public extern static void Sleep(uint msec);
  [ DllImport("kernel32.dll", EntryPoint = "Sleep") ]
  public extern static void Doze(uint msec);
  [ DllImport("user32.dll") ]
  public extern static uint MessageBox(int hwnd, String m,
                                       String c, uint flags);
  [
    DllImport("user32.dll", EntryPoint="MessageBoxW",
              ExactSpelling=true, CharSet=CharSet.Unicode)
  ]
  public extern static uint UniBox(int hwnd, String m,
                                   String c, uint flags);
}
```

Figure 19.3: Using DllImport

When calling methods that take strings, one needs to set the Unicode/ANSI policy either on the method or on the surrounding type. This is needed to control how string types are translated for consumption by unmanaged code. The `CharSet` parameter to `DllImport` allows you to specify whether Unicode (`CharSet.Unicode`) or ANSI (`CharSet.Ansi` should always be used, or if the underlying platform should automatically decide based on Windows NT/2000 vs. Windows 9x/ME (`CharSet.Auto`). Using `CharSet.Auto` is similar to writing Win32/C code using the `TCHAR` data type, except that the actual character type and API is determined at load-time, not compile-time, allowing a single binary to work properly and efficiently on all versions of Windows.

The Windows platform has a variety of name mangling schemes to indicate calling convention and character sets. When the `CharSet` is set to `CharSet.Auto`, the symbolic name will automatically have a "W" or "A" suffix, depending on whether Unicode or ANSI is used by the runtime. Additionally, if the plain symbol is not found, the runtime will munge the symbol using the `stdcall` conventions (e.g., `Sleep` may be `_Sleep@4`. This symbolic mangling can be suppressed using the `ExactSpelling` parameter to the `DllImport` attribute.

Finally, when calling Win32 functions that use COM-style `HRESULT`s, there are two options. By default, P/Invoke treats the `HRESULT` as simply a 32 bit integer that is returned from the function, requiring the programmer to manually test for failure. A more convenient way to call such a function is to pass the `PreserveSig=false` parameter to the `DllImport` attribute. This tells the P/Invoke layer to treat that 32 bit integer as a COM `HRESULT` and to

throw a `COMException` in the face of a failed result. Given the two declarations shown in figure 19.4, calling `OLE32Wrapper.CoSomeAPI1` requires the programmer to manually check the result and deal with failed HRESULTs explicitly. Because the `PreserveSig` parameter was used, calling `OLE32Wrapper.CoSomeAPI2` tells the CLR to map failed `HRESULTs` to `COMExceptions` implicitly without programmer intervention. In the case of `OLE32Wrapper.CoSomeAPI2`, the method returns a `short` that corresponds to the underlying function's final `[out,retval]` parameter. Had the P/Invoke method been declared to return `void`, then the P/Invoke layer would have assumed that the parameter list specified matches the underlying native definition exactly. This mapping only takes place when the `PreserveSig` parameter is false.

```
using System.Runtime.InteropServices;

// both functions assume the following global function:
// HRESULT CoSomeAPI([in] long a1, [out,retval] short
*pa2);

public class OLE32Wrapper {

// returns HRESULT as function result
  [ DllImport("ole32.dll", EntryPoint="CoSomeAPI") ]
  public extern static int CoSomeAPI1(int a1, out short
a2);

// throws COMException on failed HRESULT
  [ DllImport("ole32.dll", EntryPoint="CoSomeAPI",
              PreserveSig=false)  ]
  public extern static short CoSomeAPI2(int a1);
}
```

Figure 19.4: Using DllImport with PreserveSig

Typed transitions

Calling out of the runtime requires a typed method signature

- Parameters must all be typed (as with all methods in CLR)
- Isomorphic types require no conversion at transition
- Non-isomorphic types usually require conversion at transition
- Can tailor conversion using `MarshalAs` attribute

When calling out of (or into) the runtime, parameters are invariably passed on the call stack, as shown in figure 19.5. These parameters are instances of types both to the runtime and to the outside world. The key to understanding how interop works is to understand that any given "value" has two types: a managed type and an unmanaged type. More importantly, some managed types are isomorphic to an unmanaged type, which means that when an instance of that type needs to be passed outside of the runtime, no conversion is necessary. However, many types are not isomorphic and require some conversion in order to come up with a representation that is suitable for the outside world.

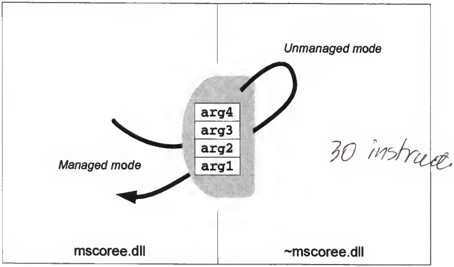


Figure 19.5: Crossing the boundary

Figure 19.6 shows a list of the basic isomorphic and non-isomorphic types. When calling an external routine that takes only isomorphic types as parameters, no conversion is needed and the caller and callee can actually share a stack frame despite the fact that one side is running in unmanaged mode. If at least one parameter is a non-isomorphic type, the stack frame must be marshaled into a format compatible with the world outside of MSCOREE. That conversion may need to take place in the other direction for parameter values passed from the function back to the caller. Fortunately, the C# compiler knows which direction parameter values flow based on the `ref` and `out` keywords. Figure 19.7 shows this effect.

Isomorphic	`Single`	*Marshals as native type*
	`Double`	
	`SByte`	
	`Byte`	
	`Int16`	
	`Uint16`	
	`Int32`	
	`Uint32`	
	`Int64`	
	`Uint64`	
	Single dimensional arrays of isomorphic types	
Non-Isomorphic	*All other arrays*	*Marshals as interface or safearray*
	`Boolean`	*VARIANT_BOOL or Win32 BOOL*
	`Char`	*Win32 WCHAR or CHAR*
	`String`	*Win32 LPWSTR/LPSTR or BSTR*
	`Object`	*VARIANT (COM Interop only)*

Figure 19.6: Isomorphic and non-isomorphic types

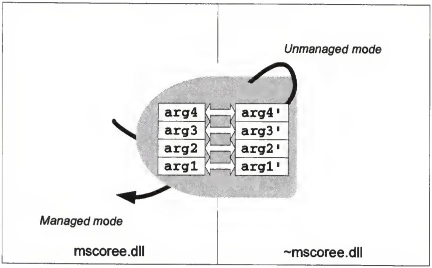


Figure 19.7: Crossing the boundary with non-isomorphic parameters

You are free to control how a given parameter (or field in a struct) marshals using the `MarshalAs` attribute. This attribute indicates which unmanaged type should be presented to the world outside MSCOREE. At the very least, one can use the `MarshalAs` attribute to indicate what the corresponding native type is for a given parameter or field. For many types, the CLR will pick a reasonable default. However, you can override these defaults using the `MarshalAs` attribute. As shown in figure 19.8, the `MarshalAs` attribute requires one parameter of type `UnmanagedType`. `UnmanagedType` is an enumerated type whose values correspond to the types the P/Invoke marshaler knows how to handle. By applying the `MarshalAs` attribute to a parameter of field, you are specifying which external type should be used by P/Invoke. Additional parameters to `MarshalAs` can be used to tailor the handling of arrays, including support for COM-style `[size_is]` using the `SizeParamIndex` parameter. Additionally, one can extend the P/Invoke marshaler by specifying a custom marshaler using the `MarshalType` parameter. This custom marshaler must implement the `ICustomMarshaler` interface, which allows the marshaler to do low-level conversions between instances of managed types and raw memory.

Parameter Name	Type	Description
Value	UnmanagedType	Unmanaged type to marshal to (mandatory)
ArraySubType	UnmanagedType	Unmanaged type of array elements
SafeArraySubType	VarType	Unmanaged VARTYPE of safearray elements
SizeConst	int	Fixed size of unmanaged array
SizeParamIndex	short	Index of parameter containing [size_is] value
MarshalType	String	Fully-qualified type name of custom marshaler
MarshalCookie	String	Cookie for custom marshaler

Figure 19.8: MarshalAs attribute parameters

The method declaration in figure 19.9 shows how the `MarshalAs` attribute can be used to map the CLR `System.String` type to one of the 4 common Win32 formats. As a point of interest, all but the `UnmanagedType.LPWStr` parameter will result in a copy of the string being created to comply with the underlying native format. Despite this, the underlying native function must never modify the buffer it is passed, as the CLR assumes that instances of `System.String` are immutable and never change. Violating this by overwriting the buffer from native code will result in a very bad situation, since the contract of `System.String` is that string objects are immutable.

```
using System.Runtime.InteropServices;

public class FooBarWrapper {

// this routine wraps a native function declared as
// void _stdcall DoIt(LPCWSTR s1, LPCSTR s2,
//                    LPTSTR  s3, BSTR s4);

  [ DllImport("foobar.dll") ]
  public static extern void DoIt(
    [MarshalAs(UnmanagedType.LPWStr)] String s1,
    [MarshalAs(UnmanagedType.LPStr)] String s2,
    [MarshalAs(UnmanagedType.LPTStr)] String s3,
    [MarshalAs(UnmanagedType.BStr)] String s4
  );
}
```

Figure 19.9: MarshalAs with parameters

In addition to using the `MarshalAs` attribute to control type mappings on a field-by-field or parameter-by-parameter basis, you can also control the underlying representation of structs and classes. In particular, the

`StructLayout` and `FieldOffset` allow you to precisely control the in-memory layout of classes and structs, which is critical for structs that are passed outside of the runtime. Figure 19.10 shows an example of an annotated C# struct, and figure 19.11 shows the equivalent definition in COM IDL.

```
using System.Runtime.InteropServices;

[ StructLayout(LayoutKind.Sequential) ]
public struct PERSON_REP {
  [ MarshalAs(UnmanagedType.BStr) ]
  public String name;
  public double age;
  [ MarshalAs(UnmanagedType.VariantBool) ]
  public bool dead;
}
```

Figure 19.10: MarshalAs

```
struct PERSON_REP {
  BSTR name;
  public double age;
  VARIANT_BOOL dead;
};
```

Figure 19.11: MarshalAs equivalent in IDL

RCW/CCW

Marshaling object references usually results in wrapper objects

- Marshaling object reference out of CLR results in a COM-callable wrapper (CCW)
- Marshaling object reference into CLR results in a Runtime-callable wrapper (RCW)
- CCW/RCW provide canned implementations of specific COM/CLR interfaces
- Lifetime issues abound due to GC vs. reference counting

When passing object references other than `System.String` or `System.Object`, the default marshaling behavior is to convert between CLR object references and COM object references. As shown in figure 19.12, when a reference to a CLR object is marshaled across the MSCOREE boundary, a COM-callable wrapper (CCW) is created to act as a "proxy" to the CLR object. Likewise, when a reference to a COM object is marshaled in through the MSCOREE boundary, a Runtime-callable wrapper (RCW) is created to act as a "proxy" to the COM object. In both cases, the "proxy" will implement all of the interfaces of the underlying object. Additionally, the "proxy" will try to map COM and CLR idioms like IDispatch, object persistence, and events to the corresponding construct in the other technology.

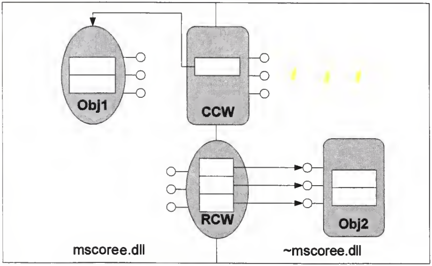


Figure 19.12: RCW/CCW architecture

For interfaces that straddle the MSCOREE boundary via RCWs or CCWs, the CLR relies on a set of annotations to the managed interface definition to give the underlying marshaling layer hints as to how to translate the types. These hints are a superset of those just described for P/Invoke. Additional aspects that need to be defined include UUIDs, vtable vs. dispatch vs. dual mappings, how IDispatch should be handled, and how arrays are translated. These aspects are added to the managed interface definition using attributes from the `System.Runtime.InteropServices` namespace. In the absence of these attributes, the CLR makes conservative guesses as to what the default settings for a given interface and method should be. For new managed interfaces that are defined from scratch, it is useful to use the attributes explicitly if you intend your interfaces to be used outside of the CLR.

TLBIMP/TLBEXP

The SDK provides tools to convert between CLR and COM type information

- `TLBEXP` converts from CLR metadata to TLB
- `TLBIMP` converts from TLB to CLR metadata
- Tons of attributes to customize the conversion
- `TLBEXP` usually does a better job than `TLBIMP`

Translating native COM type definitions (e.g., structs, interfaces, etc) to the CLR can be done by hand, and in some cases, this is necessary, especially when no accurate TLB is available. Translating type definitions in the other direction is simpler given the ubiquity of reflection in the CLR, but as always, one is better off using a tool rather than resorting to hand translations. The CLR ships with code that does a reasonable job doing this translation for you provided that COM TLBs are accurate enough. `System.Runtime.InteropServices.TypeLibConverter` can translate between TLBs and CLR assemblies. The `ConvertAssemblyToTypeLib` method reads a CLR assembly and emits a TLB containing the corresponding COM type definitions. Any hints to this translation process (e.g., `MarshalAs`) must appear as custom attributes on the interfaces, methods, fields, and parameters in the source types. The `ConvertTypeLibToAssembly` method reads a COM TLB and emits a CLR assembly containing the corresponding CLR type definitions. The SDK ships with two tools (`TLBEXP.EXE/TLBIMP.EXE`) that wrap these two calls behind a command-line interface suitable for use with NMAKE. The relationship between these two tools is shown in figure 19.13.

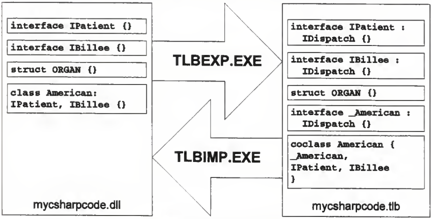


Figure 19.13: TLBIMP/TLBEXP

In general, it is easier to define types first in a CLR-based language and then emit the TLB. For example, consider the C# code shown in figure 19.14. If we were to treat this code as a "pseudo-IDL" file, we could run it through `csc.exe` and `tlbexp.exe` to produce a TLB that is functionally identical to the one produced by the "real" IDL file shown in figure 19.15. The advantage to using the C# approach is that the type definitions are extensible and easily readable, neither of which could be said for the TLB or IDL file.

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: Guid("4c5025ef-3ae4-4128-ba7b-db4fb6e0c532") ]
[assembly: AssemblyVersion("2.1") ]

namespace AcmeCorp.MathTypes {
  [
    Guid("ddc244a4-c8b3-4c20-8416-1e7d0398462a"),
    InterfaceType(ComInterfaceType.InterfaceIsIUnknown)
  ]
  public interface ICalculator
  {
    double CurrentValue { get; }
    void   Clear();

    void Add(double x);
    void Subtract(double x);
    void Multiply(double x);
    void Divide(double x);
  }
}
```

Figure 19.14: C# as a better IDL

```
[
  uuid(4C5025EF-3AE4-4128-BA7B-DB4FB6E0C532),
  version(2.1)
]
library AcmeCorp_MathTypes
{
  importlib("stdole2.tlb");
  [
    object,
    uuid(DDC244A4-C8B3-4C20-8416-1E7D0398462A),
    oleautomation,
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
           "AcmeCorp.MathTypes.ICalculator")
  ]
  interface ICalculator : IUnknown {
    [propget]
    HRESULT CurrentValue([out, retval] double* pRetVal);
    HRESULT Clear();
    HRESULT Add([in] double x);
    HRESULT Subtract([in] double x);
    HRESULT Multiply([in] double x);
    HRESULT Divide([in] double x);
  }
}
```

Figure 19.15: C# as a better IDL (generated TLB)

REGASM

The SDK provides a tool to make managed types available to COM's CoCreateInstance

- `TLBIMP` makes COM classes available to CLR via `new` operator
- `REGASM` adds COM registry entries for all CLR classes in an assembly
- MSCOREE registered as actual `InprocServer32`
- Can suppress exposure using `ComVisible` or `NoComRegistration` attributes
- Can write custom class factory if desired
- Use `REGASM /codebase` for assemblies not in GAC

Some developers want their CLR classes to be accessible via COM's `CoCreateInstance` routine. For this to happen, the proper registry entries need to be present to bind a COM CLSID to your assembly and type. The `REGASM.EXE` tool creates these registry entries by reflecting against all public types in an assembly and writing the appropriate registry keys and values. The `InprocServer32` entry always points to MSCOREE.DLL, which acts as the COM facade to your CLR-based objects. Additional registry values are written to record the assembly and type name of each interface and class under appropriate GUID. Note that the assembly name is a four-part assembly reference, not a file name, so if the assembly is not registered in the GAC, you should specify the `/codebase` command-line switch in order to also write out a code base hint into the registry.

Be aware that `REGASM.EXE` is not strictly necessary, as one can fairly easily host the CLR from native code and use the default AppDomain to load types and instantiate new instances of any class. Figure 19.16 shows an example of this using the `System.Collections.Stack` class from VB6. This can be further automated using a moniker. Surfing to `http://discuss.develop.com/archives/wa.exe?A2=ind0008&L=DOTNET&P=R63059` describes the ".NET moniker" that allows you to instantiate any CLR type using classic COM's `CoGetObject` or VB6's `GetObject`.

```
' needs to reference mscorlib.tlb and mscoree.tlb

Private Sub Form_Load()
  Dim rt As mscoree.CorRuntimeHost
  Dim unk As IUnknown
  Dim ad As mscorlib.AppDomain
  Dim s As mscorlib.Stack

  Set rt = New mscoree.CorRuntimeHost
  rt.Start
  rt.GetDefaultDomain unk
  Set ad = unk
  Set s = ad.CreateInstance("mscorlib", _

"System.Collections.Stack").Unwrap
  s.Push "Hello"
  s.Push "Goodbye"
  s.Push 42
  MsgBox s.Pop()
  MsgBox s.Pop()
  MsgBox s.Pop()
End Sub
```

Figure 19.16: Hosting the CLR from VB

COM+ 1.0

CLR classes can be exported as COM+ 1.0 configured components

- COM+ 1.0 attributes exposed as CLR attributes in `System.EnterpriseServices` namespace
- Configured CLR classes need to derive from `ServicedComponent`
- Classic `CoGetObjectContext` exposed via `ContextUtil` class
- Classic `IObjectControl/IObjectConstruct` methods are virtuals on `ServicedComponent` base type
- CLR loader auto-registers with COM+ catalog
- `REGSVCS.EXE` tool manually registers with COM+ catalog

To allow CLR code to use distributed transactions and other COM+ 1.0 services, CLR classes can be exported as COM+ 1.0 configured components. For a CLR class to be configured with the COM+ catalog, the class must derive either directly or indirectly from `System.EnterpriseServices.ServicedComponent`. This base type signals the CLR that COM+ 1.0 context will be used and alters the way the interop layer deals with the class.

The `ServicedComponent` base type dispatches incoming COM+ 1.0 `IObjectControl` and `IObjectConstruct` calls through virtual methods that derived classes can elect to override. The signature of these methods is shown in figure 19.17. CLR code can access the COM+ 1.0 context through the `System.EnterpriseServices.ContextUtil` class. As shown in figure 19.18, this class exposes static methods for each of the methods exposed by the COM+ 1.0 context object (as exposed through `CoGetObjectContext`).

```
namespace System.EnterpriseServices {
  public abstract class ServicedComponent
                  : System.ContextBoundObject {
// IObjectControl methods
    public virtual void Activate() { }
    public virtual bool CanBePooled() { return false;}
    public virtual void Deactivate() {}

// IObjectConstruct methods
    public virtual void Construct(string s) {}
  }
}
```

Figure 19.17: System.EnterpriseServices.ServicedComponent

```
namespace System.EnterpriseServices {
  public class ContextUtil {
// IObjectContextInfo methods
    public static Guid ContextId { get; }
    public static Guid ActivityId { get; }
    public static Guid TransactionId { get; }
    public static bool IsInTransaction { get; }
    public static object Transaction { get; }

// IContextState methods
    public static bool DeactivateOnReturn { get; set; }
    public static TransactionVote MyTransactionVote { get;
set; }

// IObjectContext methods
    public static bool IsSecurityEnabled { get; }
    public static bool IsCallerInRole(string role);
    public static void SetComplete();
    public static void SetAbort();
    public static void DisableCommit();
    public static void EnableCommit();
  }
}
```

Figure 19.18: System.EnterpriseServices.ContextUtil

The `System.EnterpriseServices` assembly also provides custom CLR attributes that allow you to set the initial COM+ 1.0 attribute values in your code. These attributes are used by the CLR->COM+ registration infrastructure when the component is registered with the COM+ catalog. This registration happens automatically the first time the CLR instantiates an object based on the class. You can also perform this registration manually using the `REGSVCS.EXE` tool. Figure 19.19 shows a COM+ 1.0 configured component written in C#.

```
using System;
using System.EnterpriseServices;
[assembly: ApplicationName("YetAnotherBankDemo") ]
[assembly: ApplicationActivation(ActivationOption.Library)
]

[ Transaction(TransactionOption.Required) ]
public class ReallyTiredDemo : ServicedComponent, IBank {
  void IBank.Transfer() {
    ContextUtil.SetAbort();
    DoDataAccessStuff();
    ContextUtil.SetComplete();
  }
}
```

Figure 19.19: A COM+ 1.0 configured class

Strategies

Several .NET transition strategies are possible

- Partial port of source code to a new language
- Full port of source code to new language + CLR-based libraries
- No port - use interop services to expose COM component as CLR component ("punt" approach)
- No one answer is correct for all applications

The question of "to port or to punt" is a common dilemma that many organizations face. Porting classic C++/VB6 code to support managed types requires manual labor and, due to the subtle (and not so subtle) changes in language semantics, requires an ultra-stable code base prior to the porting effort. A somewhat safer approach is to in essence "quarantine" the unmanaged code into classic Win32 or COM DLLs and use the CLR and C# for new development only. Depending on the call patterns between components, this may actually be the higher performance choice, as reducing the number of managed/unmanaged transitions is always a good thing.

There are three basic strategies for moving native code into the runtime. Figure 19.20 illustrates the partial port approach, in which source code is mechanically translated from a pre-CLR language (e.g., VB6) to a CLR-aware language (e.g., VB.NET or C#). This approach relies on the fact that any subordinate native libraries can be imported into the CLR using interop services.

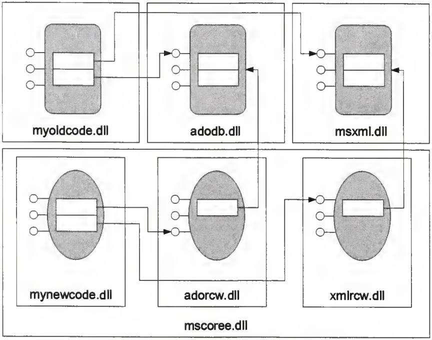


Figure 19.20: .NET via partial source-code porting

Figure 19.21 illustrates the full port approach, in which a source code is completely rewritten in a CLR-aware language without relying on any native sub-components. This approach relies on the existence of managed versions of

any subordinate native libraries and that the developer is willing to adapt their source code to any stylistic differences the managed libraries may mandate. This approach requires the most labor, but has the advantage of fewer interop transitions as well as (typically) richer/better designed libraries.

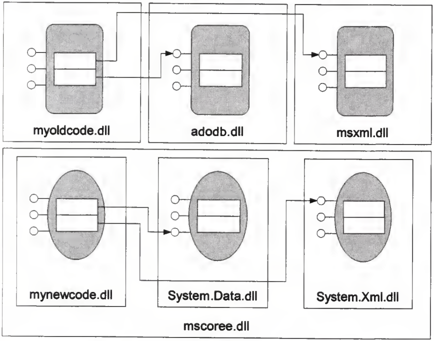


Figure 19.21: .NET via full source-code porting

Figure 19.22 illustrates the "punt" approach, in which a source code is left alone and the code is imported into the CLR using interop services. This approach does not rely on the existence of managed versions of any subordinate native libraries, nor does the developer need to adapt their source code in any significant manner. This approach requires the least labor, and may have the advantage of fewer interop transitions due to the ratio of subordinate library calls per method.

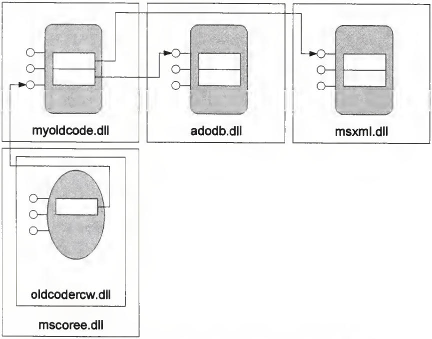


Figure 19.22: .NET via binary import (punt approach)

While these three choices aren't the only choices (one can always use things like XML or SOAP as well), it is fortunate that the CLR provides developers with a variety of choices in terms of transitioning from the old world of COM and Win32 to the new world of C# and Web Services.

Summary

- Not everything can run in MSCOREE
- Calling across the MSCOREE boundary happens at method execution
- Unmanaged DLL entry points get mapped to static methods
- Managed types can be mapped to unmanaged types automatically
- Interop relies on strongly-typed transitions
- Interop relies on extensibility points in the type information

Module 20

Appendix: Type Potpourri

Programming on the .NET platform requires literacy in a variety of issues related to the type system. Typed exceptions are used to communicate abnormal execution. Arrays and collections are two ways to build aggregations of objects types at runtime. Text processing under .NET requires C++ and VB programmers to rethink the way strings and buffers relate.

After completing this module, you should be able to:

- declare, initialize and use array variables
- effectively choose between jagged and rectangular arrays
- leverage the type compatibility rules for arrays in the face of inheritance
- write programs that use both static and dynamic text strings
- avoid excessive garbage collection when using strings
- write classes that control their formatted representation
- use the system-provided collection classes
- use the foreach construct to iterate over a collection
- write classes that act like collections
- understand the impact of boxing when using collections

Arrays

Arrays are used to manage multiple instances of a given type as a single object. Array elements are accessed by position using zero-based offsets of type System.Int32.

Arrays defined

Arrays are an aggregate type whose elements are accessed by position, not by name

- Arrays are objects
- Arrays are themselves instances of reference types
- Each language has its own syntax for array initialization and access
- The total number of elements available via the `Length` property

The CLR supports two kinds of composite types: one kind whose members are accessed by a locally unique name, the other whose members are unnamed but instead are accessed by position. The classes and structs described so far are examples of the former. Arrays are an example of the latter.

Most programming languages have some sort of array type. It is the job of the compiler to map the language-level array syntax down to a system array type. In the CLR, an array is an instance of a reference type, which means that it can implement interfaces and methods, as well as be passed where `System.Object` is expected. Independent of the language in use, the total number of elements in the array is always available using the `Length` property.

Each programming language provides its own syntax for declaring array variables, initializing arrays, and accessing array elements. Figure 20.1 shows a simple C# program that creates and uses a single-dimensional array of integers. The C# programming language supports a variety of syntaxes for initializing arrays. Figure 20.2 shows three different examples that yield identical results.

```
// declare reference to array of Int32
int[] rgn;
// allocate array of 9 elements
rgn = new int[9];
// touch all elements (index 0 through 8)
for (int i = 0; i < rgn.Length; i++)
  rgn[i] = (i + 1) * 2;
```

Figure 20.1: Creating and using an array

```
// correct
int[] a = new int[4];
for (int i = 0; i < a.Length; i++)
  a[i] = (i + 1) * 2;

// compact
int[] b = new int[] { 2, 4, 6, 8 };

// ultra-compact
int[] c = { 2, 4, 6, 8 };
```

Figure 20.2: Array initialization

Array elements

All elements of an array must be of the same type

- Arrays of value type are strictly homogeneous
- Arrays of reference type are polymorphic due to substitution
- All array elements are initialized to zero or `null`

An array consists of zero or more elements. These elements are accessed by position and must have a uniform type. For arrays of value types, each element will be an instance of exactly the same type (e.g., `System.Int32`. For arrays of reference types, each element may refer to instances of classes that support at least the element type, but may in fact be instances of derived types.

When an array is first instantiated, its elements are set to zero for value types. Figure 20.3 shows an array of value types once each element has been initialized. For arrays of reference types, each element is initially `null` and must be overwritten with a valid object reference to be useful. Figure 20.4 shows an array of reference types once each element has been initialized.

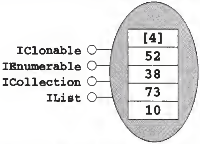


Figure 20.3: Single-dimensional array of value type

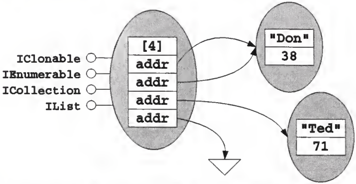


Figure 20.4: Single-dimensional array of reference type

Array capacity

An array's length is established when it is created and cannot change

- Arrays can be single or multi-dimensional
- Multi-dimensional arrays may be rectangular or C-style
- Multi-dimensional arrays may be jagged or Java-style
- Jagged arrays are not CLS-compliant

While the contents of an array can change once it has been created, the actual shape or capacity of the array is immutable and set at array-creation time. The CLR provides higher-level collection classes for dynamically-sized collections. It is interesting to note that the array capacity is not part of its type. For example, the code in figure 20.2 declares several array variables, but does not specify the capacity of the array until the `new` operator is used. This is possible because the type of an array is based only on its element type and number of dimensions (also known as rank), not its actual size.

Arrays in the CLR can be multi-dimensional. The preferred format for a multi-dimensional array is a "rectangular" array, or a C-style array. A rectangular array has all of its elements stored in a contiguous block, as shown in figure 20.5. Each "row" in a rectangular array must have the same capacity, hence the term "rectangular". Figure 20.6 shows a simple rectangular array program. Note the use of commas to delimit the index of each dimension. Also note the use of the `GetLength` method to determine the length of each dimension. For rectangular arrays, the `Length` property returns the total number of elements in all dimensions. Additionally, rectangular arrays have a variety of initialization syntaxes, which are shown in figure 20.7.

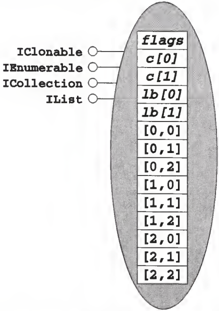


Figure 20.5: Rectangular multi-dimensional array

```
// declare reference to 2D array of Int32
int[,] matrix;
// allocate array of 3x4 elements
matrix = new int[3,4];
// touch all elements in order
for (int i = 0; i < matrix.GetLength(0); i++)
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix[i,j] = (i + 1) * (j + 1);
```

Figure 20.6: Creating and using a multi-dimensional array

```
int[,] matrix = { { 1, 2, 3, 4  },
                  { 2, 4, 6, 8  },
                  { 3, 6, 9, 12 } };
```

Figure 20.7: Multi-dimensional array initialization

Another form of multi-dimensional array is a "jagged" array, or a Java-style array. A jagged array is really just an "array of arrays" and rarely if ever has its elements stored in a contiguous block, as shown in figure 20.8. Each "row" in a jagged array may have different capacity, hence the term "jagged". Figure 20.9 shows a simple jagged array program. Note the alternate syntax for indexing each dimension. Also note the use of the `Length` property now works as expected, as the "root" array is actually a one-dimensional array whose elements are themselves arrays. While jagged arrays are quite flexible, they don't lend themselves to the same optimizations as a rectangular array. Also, jagged arrays are not CLS compliant and VB.NET has a difficult time handling them.

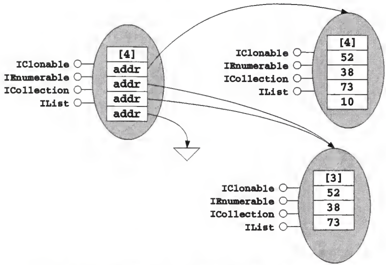


Figure 20.8: Jagged multi-dimensional array

```
// declare reference to jagged array of Int32
int[][] matrix;
// allocate array of 3 elements
matrix = new int[3][];
// allocate 3 subarrays of 4 elements
matrix[0] = new int[4];
matrix[1] = new int[4];
matrix[2] = new int[4];
// touch all elements in order
for (int i = 0; i < matrix.Length; i++)
  for (int j = 0; j < matrix[i].Length; j++)
    matrix[i][j] = (i + 1) * (j + 1);
```

Figure 20.9: Creating and using a jagged array

System.Array

Arrays are themselves reference types that extend `System.Array`

- All arrays are type-compatible with `System.Array`
- An array's type is based solely on its element type and rank
- `System.Array` utility methods support manipulation of array contents

Arrays are instances of a reference type. That reference type is "synthesized" by the CLR based on the element type and rank of the array. For example, the synthetic array type (e.g., `int[,]`) itself extends the built-in type `System.Array`, which is shown in figure 20.10. That means that all of the methods of `System.Array` are available to any type of array. That also means that one can write a method that accepts any type of array by declaring a parameter of type `System.Array`. In essence, `System.Array` identifies the subset of objects that are actually arrays.

```
namespace System {
  public class Array {
// size/shape properties
    int Length { get; }
    int Rank   { get; }
    int GetLength(int dimension);
// getters
    Object GetValue(int i);
    Object GetValue(int i, int j);
    Object GetValue(int i, int j, int k);
    Object GetValue(int [] indices);
// setters
    void SetValue(Object value, int i);
    void SetValue(Object value, int i, int j);
    void SetValue(Object value, int i, int j, int k);
    void SetValue(Object value, int [] indices);
  }
}
```

Figure 20.10: System.Array (excerpt)

Array types have their own type-compatibility rules based on the element type and the "shape" of the array. The shape of the array consists of the number of dimensions (also known as rank) as well as the capacity of each dimension. For determining type-compatibility, two arrays whose element types and rank are identical are type-compatible. Also, an array whose element type is `T` is type-compatible with all same-rank arrays with element type `V` provided `T` is type-compatible with `V`. What this means is that all single-dimensional arrays are type-compatible with the type `System.Object[]`, since all possible element types are themselves type-compatible with `System.Object`. Figure 20.11 illustrates this concept.

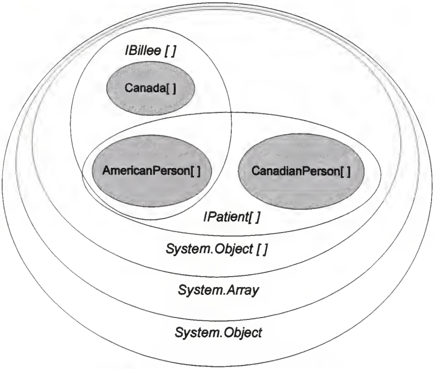


Figure 20.11: Arrays and type-compatibility

Arrays support a common set of operations. Beyond the basic accessor methods shown in figure 20.10, arrays support bulk copy operations, which are listed in figure 20.12. In particular, the `Copy` method supports copying a range of elements from one array into another. Figure 20.13 shows these methods in action.


```
namespace System {
  public class Array {
// copy descriptor + elements
    Object Clone();
// copy elements
    static void Copy(Array source,
                     Array dest,
                     int nElems);
    static void Copy(Array source,
                     int sourceOffset,
                     Array dest,
                     int sourceOffset,
                     int nElems);
// clear (zero or null) elements
    static void Clear(Array source,
                      int initialIndex,
                      int nElems);
  }
}
```

Figure 20.12: System.Array (excerpt)

```
void Main() {
  int[] left = { 1, 2, 3, 4, 5, 6 };
  Array.Clear(left, 3, 2);
// left == { 1, 2, 3, 0, 0, 6 }
  Array.Copy(left, 0, left, 3, 2);
// left == { 1, 2, 3, 1, 2, 6 }
}
```

Figure 20.13: Using System.Array

IComparable

Arrays support sorting and searching if the elements implement `System.IComparable`

- `IComparable` expresses order for a given type
- Arrays support fast binary search for ordered types
- Arrays support slow linear search for unordered types
- Arrays support forward and reverse sorts
- Ordered methods only work for single-dimensional arrays

Arrays provide extra functionality when used with ordered types. An ordered type is a type that implements `System.IComparable`. As shown in figure 20.14, `IComparable` has one method, `CompareTo`, which returns one of three values: a negative number if object's value is less than that of the argument, a positive number if object's value is greater than that of the argument, or zero if the object's value is equivalent to that of the argument. All of the primitive types are ordered and implement `IComparable`. Classes you write can be ordered provided you implement `IComparable` in a way that makes sense for your type. Figure 20.15 shows an ordered class that implements `IComparable`.

```
namespace System {
  public interface IComparable {
    int CompareTo(Object rhs);
  }
}
```

Figure 20.14: System.IComparable

```
public class Person : System.IComparable {
  double age;
  public int CompareTo(Object rhs) {
    if (this == rhs) return 0; // same
    Person other = (Person)rhs;
    if (other.age > this.age)
      return -1;
    else if (other.age < this.age)
      return 1;
    else
      return 0;
  }
}
```

Figure 20.15: Implementing System.IComparable

`System.Array` has several methods that only apply when the elements support `IComparable`. These methods are shown in figure 20.16. Technically, `Array.IndexOf` and `Array.LastIndexOf` only require the elements to implement `Equals` in a meaningful way. Figure 20.17 shows both the `IndexOf` and `BinarySearch` in action. The `BinarySearch` method requires the array to be pre-sorted, however, it performs in O(log(n)) time, which is considerably better than the O(n) time taken by `IndexOf`.


```
namespace System {
  public class Array {
// linear search using Object.Equals (-1 if not found)
    static int  IndexOf(Array array, Object value);
    static int  LastIndexOf(Array array, Object value);
// sort elements in place using IComparable
    static void Sort(Array array);
// reverse sort elements in place using IComparable
    static void Reverse(Array array);
// binary search a sorted array
// (negative index of next highest value if not found)
    static int  BinarySearch(Array array, Object value);
  }
}
```

Figure 20.16: System.Array (excerpt)

```
void Main() {
  int[] values = { 0, 2, 4, 6, 8, 10, 12 };
  int index = Array.IndexOf(values, 6); // returns 3
  index = Array.BinarySearch(values, 6); // returns 3
  index = Array.IndexOf(values, 7); // returns -1
  index = Array.BinarySearch(values, 7); // returns -4
}
```

Figure 20.17: Using System.Array revisited

Text Basics

Most applications use text extensively. The CLR provides a variety of ways of dealing with text as well as converting objects into textual representations.

System.String

All strings in the runtime are instances of the class System.String

- `System.String` objects are immutable and cannot be changed
- `System.String` objects can be crufted up from other strings
- `System.String` objects can be compared
- The runtime internalizes string literals to reduce bloat and speed up comparison
- `System.String` provides tons of convenience methods

All string values in the runtime are represented as full-blown objects that support interfaces and methods. These string objects are instances of the class `System.String`, which is the underlying type of the C# language type `string`. All string literals in programs are converted at load-time into string objects, and strings can be constructed dynamically in a number of ways.

Figure 20.18 shows the basic constructors, properties and methods of the `System.String` class. A string object can be programmatically constructed from either an array of `chars` or using a repeated sequence of chars. Once a string is constructed, it can be treated as a read-only array of `chars` using the `Length` property as well as the indexer property. It is important to note that while it is possible to read individual characters, there is no way to set individual characters. This is because objects of type `System.String` are immutable and cannot be changed. The runtime relies on the static nature of strings to avoid excessive copying when doing text processing. It is also important to note that the `char` and `System.String` types deal in Unicode characters, in particular the UTF-16 encoding scheme. This means that each string requires 2 bytes per character plus a fixed overhead of approximately 16 bytes.

```
namespace System {
  public sealed class String
         : IComparable, ICloneable, IConvertable {
// constructors
    public          String(char[] value);
    public          String(char[] value,
                           int offset, int length);
    public          String(char c, int count);
// array-like properties
    public int      Length { get; }
    public char     this[int index] { get;  }
    public char[] ToCharArray();
    public char[] ToCharArray(int offset, int length);
// obvious operations
    public static String Concat(String s1, String s2);
    public String Substring(int offset);
    public String Substring(int offset, int length);
  }
}
```

Figure 20.18: System.String (excerpt)

Figure 20.18 shows some of the basic operations one can perform on the `System.String` class. Note that programming languages may provide operators (`+` in C#, `&` in VB.NET) for aliasing calls to `String.Concat`. Additionally, most compilers are smart enough to fold the catenation of two

string literals into a single string. That means that the expression `"Hello, " + "World"` will be compiled down to the same code as `"Hello, World"`, not `String.Concat("Hello, ", "World")`. Figure 20.19 shows some simple text manipulation.

```
void Main() {
  String s1 = "Hello";
  String s2 = s1.Substring(0, 4);
  String s3 = String.Concat(s1,
                      String.Concat(" ", s2));
  bool truth = s3.Length == 10;
  truth = s3[4] == 'o';
  truth = s3 == "Hello Hell";
}
```

Figure 20.19: Simple string manipulation

When comparing strings, the `String.Equals` method has been implemented to compare the values of the two strings character by character. Additionally, the C# programming language overloads the == operator to compare for equivalence, not identity. That means that to compare two strings for identity, once must first cast the two strings to `System.Object` to defeat C#'s overloading, as shown in figure 20.20.

```
void f(String s1, String s2) {
// perform case-insensitive deep
// comparison for lt/gt/eq
  int cmp = String.Compare(s1, s2, true);
// perform deep comparison
  bool sameValue = s1.Equals(s2);
  sameValue = s1 == s2; // overloaded in C# to Equals
// perform shallow/reference comparison
  bool sameObject = ((object)s1 == (object)s2);
// compare internalized strings
  String is1 = String.Intern(s1);
  String is2 = String.Intern(s2);
  sameValue = ((object)is1 == (object)is2);
}
```

Figure 20.20: String comparison

Beyond the basic functionality already described, the `System.String` class provides a non-trivial number of convenience methods for tokenizing, parsing, cracking and generally munging of string objects. Most of these methods are shown in figure 20.21.


```
namespace System {
  public sealed class String
          : IConvertable, IComparable, ICloneable,
         System.Collections.IEnumerable {
    public bool     StartsWith(String suffix);
    public bool     EndsWith(String suffix);
    public int      IndexOf(String substr);
    public int      IndexOf(char c);
    public String   ToUpper();
    public String   ToLower();
    public String   Trim(char[] fuzz);
    public String   TrimStart(char[] fuzz);
    public String   TrimEnd(char[] fuzz);
    public String[] Split(char[] separators);
    public String   Join(String separator, String[]
tokens);
  }
}
```

Figure 20.21: System.String miscellany

ToString

Any object can be converted to a string via the `ToString` method

- Default behavior returns the namespace-qualified typename
- Can override default behavior to return whatever you want
- `ToString` is the quick-and-dirty object-to-string conversion

Any object can be converted to a string via `System.Object`'s `ToString` virtual method. The default implementation of `ToString` is to return the namespace-qualified type name of the object. Objects that care about this conversion may provide their own implementation, as shown in figure 20.22. Note that this code relies on the built-in types implementing `ToString` in a meaningful way (which they do)

```
public class Person {
  String name;
  int    age;
  public override String ToString() {
    return String.Concat(name, ": ", age.ToString());
  }
}
```

Figure 20.22: Implementing ToString

System.Text.StringBuilder

`System.Text.StringBuilder` is for strings that can change

- Supports basic insert/append/delete operations
- Internally maintains a dynamic read/write buffer of characters
- Use of `System.Text.StringBuilder` reduces GC overhead considerably

Excessive use of string catenation can kill performance both by swamping the garbage collector with extraneous heap maintenance, but also in terms of resource consumption and memory movement costs. To avoid these performance problems, the CLR provides the `System.Text.StringBuilder` class. Figure 20.23 shows the basic members of `System.Text.StringBuilder`, which includes support for simple insert/append/remove operations on a dynamic string. As shown in figure 20.24, the `System.Text.StringBuilder` internally maintains a read/write resizable character buffer (which is layout compatible with `System.String`. An example of `System.Text.StringBuilder` in action is shown in figure 20.25.

```
namespace System.Text {
  public sealed class StringBuilder {
// constructors
    public          StringBuilder();
    public          StringBuilder(String initValue);
    public          StringBuilder(int initCapacity,
                                  int maxCapacity);
// properties
    public int      Length { get; }
    public int      Capacity { get; }
    public int      MaxCapacity { get; }
    public char     this[int index] { get;  set; }
// operations
    public StringBuilder Append(String value);
    public StringBuilder Insert(int offset, String value);
    public StringBuilder Remove(int offset, int length);
    public StringBuilder Replace(String oldVal,
                                 String newVal);
    public StringBuilder Replace(char   oldVal,
                                 char   newVal);
    public String        ToString(int offset, int length);
  }
}
```

Figure 20.23: System.Text.StringBuilder (excerpt)

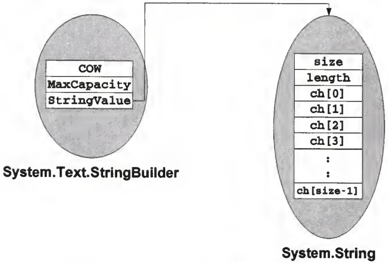


Figure 20.24: System.Text.StringBuilder

```
void Main() {
  StringBuilder sb = new StringBuilder(16, 100);
  sb.Append("Hello").Append(" ").Append("Hell");
  sb[6] = 'Y';
  sb.Insert(6, "to ");
  bool truth = sb.ToString() == "Hello to Yell";
}
```

Figure 20.25: Simple string manipulation using System.Text.StringBuilder

String.Format

`String.Format` supports flexible formatting of objects into strings

- Allows string-ified objects to be interspersed with literal text
- Relies on `ToString` method to convert objects to text
- Used by `System.Console.WriteLine`

The `System.String` class provides an extremely useful static method called `Format`, the various overloads of which are shown in figure 20.26. In each of these overloads, the first parameter is a format string that contains both literal text as well as object placeholders (which are denoted using `{i}` where `i` is the zero-based index of the placeholder. Figure 20.27 shows the syntax for a placeholder.

```
namespace System {
  public sealed class String
    : IConvertable, IComparable, ICloneable {
    public static String Format(String format,
                                Object val0);
    public static String Format(String format,
                                Object val0,
                                Object val1);
    public static String Format(String format,
                                Object val0,
                                Object val1,
                                Object val2);
    public static String Format(String format,
                                Object [] vals);
    public static String Format(IFormatProvider fp,
                                String format,
                                Object [] vals);
  }
}
```

Figure 20.26: System.String.Format

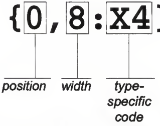


Figure 20.27: Format string placeholder syntax

The `String.Format` method processes the format string and replaces the placeholders with the string-ified version of the associated parameter. For example, the expression `String.Format("{0}, {1}, {0}", "Hello", "World")` would return the string `"Hello, World, Hello"`.

Note that the arguments after the format string do not have to be strings themselves (they are typed as `System.Object`. By default, the implementation of `String.Format` simply calls `ToString` on each argument when it needs the string to replace the placeholder with. That means that the expression `String.Format("{0}, {1}, {0}", 8, 3.1)` would return the string `"8, 3.1, 8"`.

It is also possible to control the width of each placeholder using the optional field-width argument. If the field-width argument is positive, the placeholder will be replaced with a right-justified string (space-padded). If the field-width argument is negative, the placeholder will be replaced with a left-justified string (space-padded). For example, the expression `String.Format("{0,2}{1,-2}", 8,9)` would return the string `" 89 "`.

While `String.Format` is an extremely useful method, most programmers' first exposure to it is via the `System.Console.WriteLine` method, which simply calls `String.Format` prior to writing the text to the console.

System.IFormattable

Objects can support their own custom formats by implementing `System.IFormattable`

- Numeric types have a rich set of codes predefined
- You can define your own for your own classes
- Also possible to provide out-of-band converter for built-in types

The `String.Format` method supports type-specific formatting codes. These formatting codes come after the placeholder position and field-width and are separated by a colon, as shown in figure 20.27. Figure 20.28 shows the type codes for the built-in numeric types. Note that for each type code, you may provide a numeric suffix, which influences either leading or trailing zeros. For example, the expression `String.Format("0x{0:X8}", 57005)` would return the string `"0x0000DEAD"`. Had a lower-case "x" been used instead of an upper-case "X", the resultant string would have been `"0x0000dead"`. The numeric types also support picture-style format strings a la products such as Microsoft Excel. These picture codes are shown in figure 20.29.

Code	Meaning	Positive	Negative	Suffix
`G`	General	`12345.6789`	`-12345.6789`	Ignored
`C`	Currency	`$12,345.68`	`($12,345.68)`	Trailing Zeros
`E`	Scientific	`1.23456789E-004`	`-1.23456789E-004`	Trailing Zeros on Mantissa
`F`	Fixed point	`12345.68`	`-12345.68`	Trailing Zeros
`N`	Number	`12,345.68`	`-12,345.68`	Trailing Zeros
`X`	Hexadecimal*	`3039`	`FFFFCFC7`	Leading Zeros
`D`	Decimal*	`12345`	`-12345`	Leading Zeros

** Integral types only*

Figure 20.28: String.Format numeric codes

Code	Meaning
0	Zero placeholder
#	Significant digit placeholder
.	Decimal point
%	Literal percent sign
E+0	Exponent
\	Literal character
"ABC"	Literal character string
"ABC"	Literal character string
x;y;z	Selects x if positive, y if negative, z if zero

Figure 20.29: String.Format picture codes

Any class can support its own custom format codes by implementing the `System.IFormattable` interface. This interface is shown in figure 20.30 and has exactly one method: `ToString`. When `String.Format` tries to convert an object to text, it first tries to use the object's `IFormattable` interface. If the object does not support the interface but there are format codes that need to be interpreted, `String.Format` throws a `FormatException` indicating that an unrecognized format string was found. However, if the object being converted does implement `IFormattable`, the type-specific format code is passed to the object's `IFormattable.ToString` method. Only the type-specific code is passed, not the placeholder index or field width. In the previous example, the string passed to `IFormattable.ToString` would be `"X8"`. Figure 20.31 shows a typical implementation of `IFormattable`. Assuming a properly initialized variable `p` of type `Person`, the expression `String.Format("{0:N}`

`is {0:A} years old", p)` would return the string `"Don is 38 years old"`. Figure 20.32 shows exactly how a typical call to `System.Console.WriteLine` is processed.

```
namespace System {
  public interface IFormattable {
    String ToString(String format, IFormatProvider fp);
  }
  public interface IFormatProvider {
    object GetFormat(Type type);
  }
}
```

Figure 20.30: System.IFormattable

```
public class Person : IFormattable {
  String name;
  int    age;
  public String Format(String format) {
    if (format == null || format == "G")
      return this.ToString();
    if (format == "N")
      return this.name;
    else if (format == "A")
      return this.age.ToString();
    else
      throw new FormatException("Unrecognized code: "
                                + format);
  }
}
```

Figure 20.31: Implementing System.IFormattable

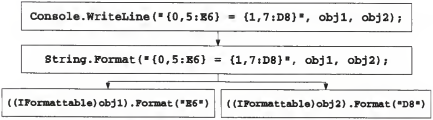


Figure 20.32: Inside Console.WriteLine

Collections

Arrays are an intrinsic collection type, but many others are possible. The System.Collections namespace defines interfaces and classes that codify the creation and use of collection types.

IEnumerable/IEnumerator

The CLR defines the `System.Collections.IEnumerable/IEnumerator` interfaces to support iterating across an arbitrary collection

- `IEnumerable` is implemented by objects that act like collections
- `IEnumerable.GetEnumerator` returns a logical "cursor" object
- `IEnumerator` supports forward traversal through a bounded collection
- `System.Array` and the standard collection classes all implement `IEnumerable`
- Implementations are free to implement these interfaces however they like

The CLR defines the a standard pair of interfaces to support iterating across arbitrary collections. `System.Collections.IEnumerable` is implemented by objects that support iteration. `System.Array` supports this interface, as do all of the built-in collection classes. As shown figure 20.33, the `IEnumerable` interface only has one method, `GetEnumerator` that allows the caller to request a new enumerator to traverse the collection with. That enumerator is simply an iterator or cursor over the underlying collection.

```
namespace System.Collections {
  public interface IEnumerable {
    IEnumerator GetEnumerator();
  }
  public interface IEnumerator {
    bool MoveNext();
    object Current { get; }
    void Reset();
  }
}
```

Figure 20.33: IEnumerable/IEnumerator

Enumerators implement the `IEnumerator` interface, which supports exactly three operations. The `Reset` method sets the internal cursor to refer to the initial element in the collection. The `MoveNext` method advances the internal cursor by one element, returning `false` if the cursor has gone off the end of the collection. The `Current` property dereferences the cursor and returns a reference to the element underneath the internal cursor. Figure 20.34 shows a simple program that uses `IEnumerable/IEnumerator` to traverse a collection.

```
void g(IEnumerable collection) {
  IEnumerator e = collection.GetEnumerator();
  while (e.MoveNext()) {
    ElemType value = (ElemType)e.Current;
    value.dosomething();
  }
}
```

Figure 20.34: Using IEnumerable/IEnumerator

Figure 20.35 shows the relationship between the enumerator and the underlying collection. Note that this figure is approximate. The underlying collection does not have to physically exist in memory all at once. Additionally, the enumerator may need additional information to keep track of its position. However, most enumerators maintain at least a reference to the "top" of the

collection as well as a reference (or index) to the "current" element in the collection.

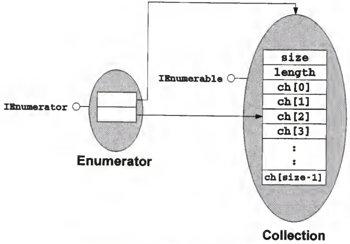


Figure 20.35: IEnumerable/IEnumerator

Programming languages and IEnumerable

Programming languages often expose constructs to simplify the use of IEnumerable

- C# and VB.NET's `foreach` writes the same code you would
- C# compiler somewhat permissive to allow better type matching

While the code shown in figure 20.34 is not that difficult to write, it is somewhat tedious. To relieve the burden on the programmer, C# and VB.NET provide language-level support for dealing with `IEnumerable/IEnumerator`. C# provides the `foreach` statement which automates the code shown in figure 20.34.

The code in figure 20.36 demonstrates the use of the `foreach` statement. This code is functionally identical to the code in figure 20.34. The `foreach` statement allows you to specify the type and name of the "current" element as well as an expression that evaluates to `IEnumerable`. The C# compiler then takes those three pieces and generates code that is identical to a hand-coded while loop.

```
void f(IEnumerable collection) {
  foreach (ElemType value in collection) {
    value.dosomething();
  }
}
```

Figure 20.36: foreach

The C# `foreach` is actually fairly forgiving. The collection expression does not necessarily need to be `IEnumerable` compatible. Rather, if the expression's type has an accessible method named `GetEnumerator` and the return type of that method has a `Current` property and a zero-parameter `MoveNext` method, then the compiler will happily treat the expression as a collection even though it doesn't support the `IEnumerable` interface. The compiler does this to allow you to write more type-safe versions of the `IEnumerable/IEnumerator` methods. For collections of value types especially, this can be a tremendous optimization, as the `IEnumerator` interface would require boxing every time the `Current` property was evaluated. Figure 20.37 shows an example of a class that is not generally enumerable but that is acceptable for use with the `foreach` statement. Figure 20.38 shows the more "legitimate" implementation.

```
public class EvenGenerator
{
  int max;
  public EvenGenerator(int max) { this.max = max; }
  public Cursor GetEnumerator()
  {
    return new Cursor(max);
  }
  public class Cursor
  {
    int max;
    int current = -2;
    internal Cursor(int max) { this.max = max; }
    public int Current { get { return current; } }
    public bool MoveNext()
    {
      if (current > max)
        return false;
      current += 2;
      return true;
    }
  }

}
```

Figure 20.37: A pseudo-implementation of IEnumerable

```
public class EvenGenerator : IEnumerable
{
  int max;
  public EvenGenerator(int max) { this.max = max; }
  public IEnumerator GetEnumerator()
  {
    return new Cursor(max);
  }
  public class Cursor : IEnumerator
  {
    int max;
    int current = -2;
    internal Cursor(int max) { this.max = max; }
    public Object Current { get { return current; } }
    public void Reset() { current = -1; }
    public bool MoveNext()
    {
      if (current > max)
        return false;
      current += 2;
      return true;
    }
  }
}
```

Figure 20.38: A plain vanilla implementation of IEnumerable

ICollection and friends

ICollection/IList/IDictionary abstract common collection types

- `ICollection` requires countability and clone-to-array support
- `IList` requires positional access/insertion/removal
- `IDictionary` requires access/insertion/removal based on unique keys

The CLR defines interfaces that restrict the set of enumerable objects into subsets based on how much additional functionality can be expected. Figure 20.39 shows how the three interfaces `ICollection`, `IList`, and `IDictionary` partition the space of all enumerable objects.

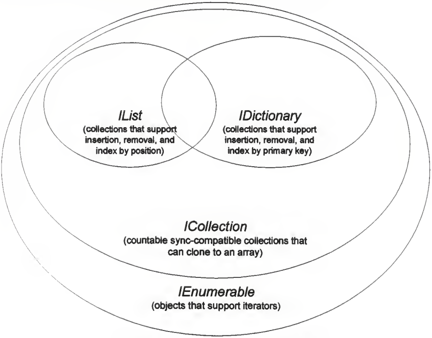


Figure 20.39: Collection interfaces

Figure 20.40 shows the definition of `ICollection`. The `ICollection` interface is implemented by enumerable objects that can report their element count. These objects also must be able to clone their contents to an array via the `CopyTo` method. Additionally, objects that implement `ICollection` may provide a thread-synchronized object to act as a lock for multi-threaded code. This object is accessible via the `SyncRoot` property and may or may not be the same object as the collection itself.

```
namespace System.Collections {
  public interface ICollection : IEnumerable {
    int    Count { get; }
    bool   IsSynchronized { get; }
    object SyncRoot { get; }

    void   CopyTo(System.Array array, int index);
  }
}
```

Figure 20.40: ICollection

Figure 20.41 shows the definition of `IList`. The `IList` interface is implemented by collections that support positional access, insertion and removal such as a dynamically sized array. Figure 20.42 shows the `IList` interface in action.

```
namespace System.Collections {
  public interface IList : ICollection {
    bool   IsFixedSize { get; }
    bool   IsReadOnly { get; }
    object this[int index] { get; set; }

    void Add(object val);
    void Clear();
    bool Contains(object val);
    int  IndexOf(object val);
    void Insert(int index, object val);
    void Remove(object val);
    void RemoveAt(int index);
  }
}
```

Figure 20.41: IList

```
static void FunWithTimKeithAndChris(IList list) {
  list.Add("Tim");
  list.Add("Keith");
  list.Add("Chris");
  foreach (object o in list)
    Console.WriteLine(o);
  for (int i = 0; i < list.Count; i++)
    Console.WriteLine(list[i]);
}
```

Figure 20.42: Using IList

Figure 20.43 shows the definition of `IDictionary`. The `IDictionary` interface is implemented by collections that support access, insertion and removal based on primary keys. Figure 20.44 shows the `IDictionary` interface in action. Note that one entry is kept for "Don" and one entry is kept for "Ted" no matter how many times values are added with those keys.

```
namespace System.Collections {
  public interface IDictionary : ICollection {
    bool   IsFixedSize { get; }
    bool   IsReadOnly { get; }
    ICollection            Keys { get; }
    ICollection            Values { get; }
    object                 this[object key] { get; set; }

    void                   Add(object key, object val);
    void                   Clear();
    void                   Remove(object key);
    bool                   Contains(object key);
    IDictionaryEnumerator GetEnumerator();
  }
  public interface IDictionaryEnumerator : IEnumerator {
    object                 Key { get; }
    object                 Value { get; }
    DictionaryEntry        Entry { get; }
  }
  public struct DictionaryEntry {
    public object Key;
    public object Value;
  }
}
```

Figure 20.43: IDictionary

```
static void FunWithDonAndTed(IDictionary d) {
  d.Add("Don", 38);
  Console.WriteLine(d["Don"]);
  d.Add("Ted", 19);
  d["Tim"] = ((int)d["Don"])/4;
  foreach (DictionaryEntry de in d)
    Console.WriteLine("{0}: {1}", de.Key, de.Value);
}
```

Figure 20.44: Using IDictionary

ArrayList and friends

The .NET Framework provides a handful of generic collection implementations

- `ArrayList` is a dynamically sized array
- `Hashtable` is a property bag
- `SortedList` is a sorted version of `HashTable`
- `Stack` is a dynamically sized first-in-last-out array
- `Queue` is a dynamically sized first-in-first-out array
- `BitArray` is a dynamically sized array of bits/booleans

In addition to the `IEnumerable` family of abstract interfaces, the .NET framework provides a handful of generic collection implementations. The catalog of built in collection classes is shown in figure 20.45. Their relationship to the abstract collection interfaces is shown in figure 20.46.

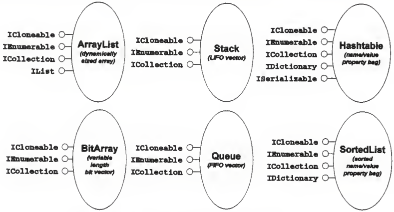


Figure 20.45: Built-in collection types

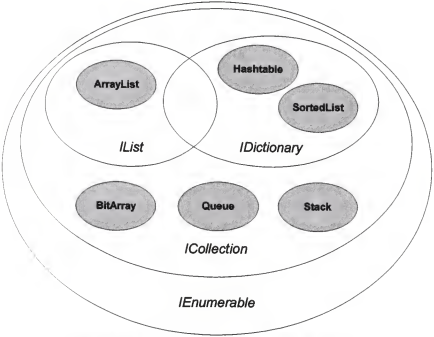


Figure 20.46: Built-in collection types (revisited)

Summary

- Arrays are types whose elements are accessed by position, not name
- Text handling is based on constructing immutable strings
- Enumeration is modeled using the `IEnumerable`/`IEnumerator` interfaces
- Collections are modeled using the `IList`/`IDictionary` interfaces

Glossary

AppDomain The basic scope of execution in the CLR.

Array An entity whose fields are accessed by position, not by name

ASP An ISAPI extension that supports generalizing text-based HTTP data using script and COM components.

ASP.NET The latest generation of Active Server Pages. The ASP.NET framework is a complete re-write of classic ASP based on .NET. Some of the significant new features include precompiled code, code behind, object oriented class libraries, and user controls.

Assembly A collection of type definitions that are deployed as a unit.

Assembly loader The core piece of code in the CLR that maps assemblies into memory prior to the use of a type.

Assembly reference A reference to an external assembly, typically found in an assembly manifest.

Assembly resolver The core piece of code in the CLR that locates assemblies by name prior to invoking the assembly loader to load them.

BASIC A friendly programming language that exemplified the "anyone can program" philosophy. BASIC depended on libraries and an interpreter for portability, and most BASIC development environments were integrated into the OS (e.g., GW-BASIC). The BASIC programming language was untyped and tried to use as few principles and constructs as possible.

C A systems programming language from Bell Laboratories that exemplified the "portable assembly language" philosophy. C depended on libraries for portability, and most C development environments generated raw machine code that ran extremely fast. The C programming language was barely typed, in fact, the original version didn't mandate typed function prototypes.

C++ A systems and applications programming language from Bell Laboratories that exemplified the "portable assembly language with object orientation" philosophy. C++ depended on libraries for portability, and most C development environments generated raw machine code that ran extremely fast. The C++ programming language was strongly typed at compile-time, however, very little runtime type information survives the build process.

CGI The original extensibility model for HTTP servers on UNIX. CGI is based on a process-per-request model and is (begrudgingly) supported by IIS.

COM A type-oriented integration technology from Microsoft that exemplified the "interface-based programming" philosophy. COM depended on IDL for portability, and most COM development environments generated raw machine code that ran extremely fast. COM added runtime type information

to the C++ object model and was adapted to virtually all Windows programming environments.

COM+ The Windows 2000 version of MTS. COM+'s primary contribution to MTS was that it integrated the MTS and COM programming models.

Common Language Runtime (CLR) A runtime system designed to support multi-paradigm programming across component boundaries. The CLR's primary feature is a pervasive type system that is extensible and easily programmed from most languages.

Delegate An object that exists solely to invoke a specific method on a target object

Event A pseudo-property of delegate type that supports registering delegates as "event handlers"

Event handler A delegate object registered with a CLR event

Field A typed, named unit of storage associated with a type or an instance of a type, depending on scope.

Finalization The act of notifying an object that its underlying memory is about to be recycled.

Garbage Collection An approach to resource reclamation based on asynchronous detection of unused resources.

Global assembly An assembly that is deployed for machine-wide use.

Hard thread An operating system thread.

IIS Microsoft's extensible HTTP server. Bundled with all versions of Windows NT/2000 and available for Windows 9x as well.

Indexer A construct that binds a type to one or two method definitions that simulate array element access

Instance A piece of memory that is affiliated with the type it is an instance of.

Instance-scope Tied to a particular instance of a type.

Intermediate Language (IL) The intermediate language generated by compilers of CLR-compliant languages in .NET. Assemblies consist of intermediate language code that is just-in-time compiled into machine code on first access.

ISAPI Microsoft's preferred extensibility model for IIS. ISAPI is based on mapping HTTP requests to user-defined DLLs based on file suffixes.

Jagged Array An array of arrays, each of which may have differing capacity

Java An object-oriented language from Sun Microsystems that married the semantics and portability of Smalltalk with the syntax of C++. Java used a supporting virtual machine for platform portability, and an entire platform's worth of supporting libraries targeted the Java virtual machine. Like Smalltalk, complete runtime type information was a given in the Java system.

JSP A Java-based homage to ASP.

Loader A piece of code that locates and prepares chunks of code for execution.

Member A field or method of a type. Properties, indexers, events, and nested typees can also be members of a type.

Method A typed, named operation associated with a type or an instance of a type, depending on scope.

Module A file that contains some or all of the type definitions in one assembly.

MSCOREE A execution engine of the CLR. MSCOREE is responsible for loading and managing types.

MTS A type-oriented extension to COM from Microsoft that exemplified the "attribute-based programming" philosophy. MTS extended the COM loader in Windows NT 4.0 to support custom attributes that would influence the execution semantics of the loaded type. MTS added context to the COM object model and was subsequently integrated into IIS.

Originator See Public Key

Overload To redefine the meaning of a name in a different context.

Primitive type A value type whose value is atomic and is known a priori by the runtime to be one of the 12 intrinsic types (System.Boolean, System.Char, System.Double, System.Single, System.SByte, System.Int16, System.Int32, System.Int64, System.Byte, System.UInt16, System.UInt32, and System.UInt64).

Private assembly An assembly that is deployed for the exclusive use of one or more applications.

Property A construct that binds a name and a type to one or two method definitions that simulate field access

Public Key An cryptographically unique and large number used to identify a user or organization. Used in the CLR to identify the developer of an assembly.

Public Key Token An cryptographic hash of a public key that facilitates fast lookups and comparisons of public keys. Used in the CLR to reduce the size of a fully-qualified assembly reference.

Rectangular Array A multi-dimensional array whose row and column size is uniform and whose members are allocated as one contiguous block of memory

Reference type A type whose variables are (potentially null) references to objects, not values. All types that do not derive from System.ValueType directly or indirectly are reference types.

Servlet A Java-based HTTP framework that functionally sat between ISAPI and ASP.

Shadow copying A feature of the assembly resolver in which a copy of the assembly's files are loaded to avoid taking a read lock on the underlying files.

Smalltalk An object-oriented language from Xerox PARC that exemplified the "everything is an object" philosophy. Smalltalk used a supporting virtual machine for platform portability, and most Smalltalk development environments ran inside such a virtual machine. Additionally, complete runtime type information was a given in the Smalltalk system.

Soft thread A CLR thread object of type System.Threading.Thread

Static-scope Shared by all instances of a type.

Thread-local-storage (TLS) Data that is specific to a particular thread.

Type A reusable abstraction that defines the operations and state for instances of that type.

User control Similar in functionality to include files in classic ASP, code can be encapsulated in a user control and used throughout a web site with little additional effort.

Value type A type whose variables are actual values, not references to objects. All value types derive from System.ValueType either directly or indirectly.

Visual Basic A friendly programming language that made Windows (and later Web) programming accessible to the masses. Visual Basic depended on a runtime (MSVBVM60.DLL), and was integrated into several popular applications via Visual Basic for Applications (VBA). Visual Basic evolved into a fairly COM-centric programming language by the end of its development in 1998.

WAM The per-web application object that bridges the world of IIS and the world of MTS/COM+.

Web Service An application that reveals itself via the Internet using SOAP as its communication protocol for method invocation. Web services can expose functionality that can be consumed by both web applications and desktop applications using SOAP, an XML-based protocol for method invocation.

WebForm An ASP.NET page containing embedded server-side controls.

XML The extensible markup language is a text-based data representation format.

Bibliography

Web Sites

ASP.NET Directory of resources: http://www.123aspx.com
ASPfree.com: http://aspfree.com
DevelopMentor DOTNET mailing list: http://discuss.develop.com/dotnet.html
IBuySpy - sample application: http://www.ibuyspy.com
Microsoft .NET home page: http://msdn.microsoft.com/net/
MSDN Magazine online: http://msdn.microsoft.com/msdnmag
Scott Guthrie's sample site: http://www.eraserver.net/scottgu/
Superexpert Controls - ASP.NET control site: http://superexpertcontrols.com
The Complete ASP.NET FAQ: http://www.aspnetfaq.com

Books

Don Box, Aaron Skonnard, John Lam. 1999. *Essential XML*. Boston MA: Addison Wesley Longman.
Tim Lindholm and Frank Yellin. 1999. *The Java Virtual Machine Specification (2nd Edition)*. Boston MA: Addison Wesley Longman.
John R Levine. 1999. *Linkers and Loaders*. San Francisco CA: Morgan Kaufmann.
Richard Anderson, Alex Homer, Rob Howard, Dave Sussman. 2000. *A Preview of Active Server Pages+*. USA: Wrox Press.
Keith Brown. 2000. *Programming Windows Security*. Boston MA: Addison Wesley Longman.

Produced by IKON Office Solutions
Outsourcing Division (503) 469-9141